TigerBeetle

for the rest of us.

Frank Denis — Fastly

Disclaimer

- I’m new to TigerBeetle
- I know nothing about finance
- But I wanted to share my experience as a user

What "Database for Finance" Really Means

When you hear this, your first instinct might be skepticism:

- Something complicated
- Something heavy and bloated
- Something expensive
- Something you don’t need unless you work for a bank

But "finance" implies something else: Toughness

Financial systems must be tough because the consequences of failure are severe.

What "Tough" Means

Reliability

- Data must never be corrupted
- Every cent must be accounted for
- Immutable audit trail

Strict Consistency

- No eventual consistency
- Same balance everywhere, always

Resilience

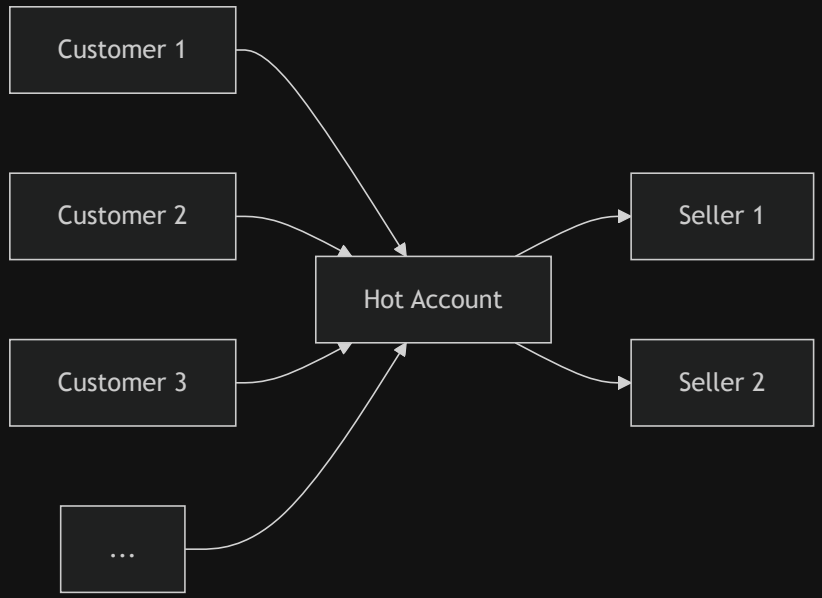
- No single point of failure
- Distributed across locations
- Keeps running when nodes fail

Performance

- Low latency under load
- Handles traffic spikes
- Scales with data growth

The Hot Account Problem

Imagine an intermediary account that processes transactions for thousands of sellers:



Every transaction touches that one account.

In traditional databases, this creates a serialization bottleneck.

Why Scaling Doesn’t Help

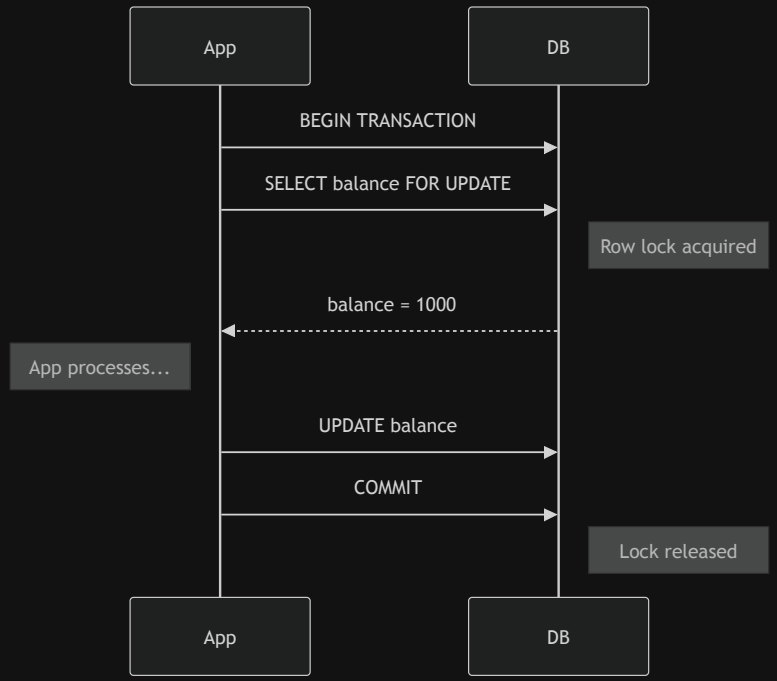
- More machines = more synchronization overhead
- More machines = more coordination
- More machines = more locking

The fundamental problem isn’t capacity - it’s contention.

Sharding doesn’t help when every transaction needs to touch the same hot account.

The Lock Problem

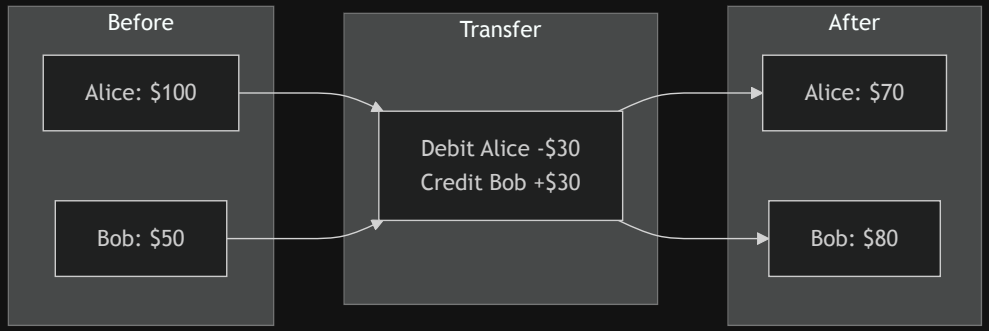
When using SQL with traditional client-server architecture:



Row locks are held during network round trips!

Every Transfer: Debit = Credit

When Alice pays Bob $30, the total money in the system stays the same:



Debits = Credits, always. Money doesn't appear or disappear - it just moves between accounts.

This Is Double-Entry Bookkeeping

Every transaction has two sides that must balance:

Account	Debit (−)	Credit (+)
Alice	$30	
Bob		$30
Total	**$30**	**$30**

This 700-year-old principle is the foundation of all financial systems. TigerBeetle enforces it automatically.

TigerBeetle's Approach

What if we designed a database to specialize in exactly this?

- **Fixed schema**: 128-byte transfers (2 cache lines)
- **Batching**: Up to 8,000 transfers per request
- **No locks across network**: Entire batch processed internally
- **Single round trip**: Send batch, get acknowledgment

At its core, it's just counting. Debit one account, credit another.

The Transfer Record

```
Transfer (128 bytes):
├── id:              u128 (16 bytes)
├── debit_account_id: u128 (16 bytes)
├── credit_account_id:u128 (16 bytes)
├── amount:          u128 (16 bytes)
├── pending_id:      u128 (16 bytes)
├── user_data_128:   u128 (16 bytes)
├── user_data_64:    u64  (8 bytes)
├── user_data_32:    u32  (4 bytes)
├── timeout:         u32  (4 bytes)
├── ledger:          u32  (4 bytes)
├── code:            u16  (2 bytes)
├── flags:           u16  (2 bytes)
└── timestamp:       u64  (8 bytes)
```

Two CPU cache lines, perfectly aligned.

Consensus: Viewstamped Replication

TigerBeetle uses **VSR** with flexible quorums:

Operation	Quorum (6 replicas)
Replication (commit)	3 replicas
View-change	4 replicas
NACK (truncation)	4 replicas

Can tolerate **3 replica failures** (50% of cluster) and remain available.

$20,000 bounty for finding bugs: github.com/tigerbeetle/viewstamped-replication-made-famous

Protocol-Aware Recovery

Disk corruption is detected and repaired transparently:

- **Checksums** on every data unit (blocks, prepares, requests)
- **Physical repair**: byte-for-byte identical replication
- **No tree retransmission**: if a single block corrupts, only that block is repaired

Deterministic execution means any healthy replica can repair any corrupted replica to an identical state.

Tiger Style

Engineering practices inspired by NASA's "Power of Ten":

- **Static memory allocation** - all memory pre-allocated at startup
- **No recursion** - bounded execution guaranteed
- **Hard limits on everything** - loops, queues, buffers, batch sizes
- **70 lines per function** - hard limit, no exceptions
- **Assert everything** - 2+ assertions per function average

A crash is recoverable; silent data corruption is not.

Tiger Style (cont.)

- **Zero dependencies** - only Zig toolchain, no supply chain risk
- **Zero technical debt** - do it right the first time
- **Pair assertions** - every property checked from 2+ code paths
- **All errors handled** - 92% of catastrophic failures come from ignored errors
- **Optimize in order**: Network → Disk → Memory → CPU

> "Assertions downgrade catastrophic correctness bugs into liveness bugs."

Testing: The VOPR

Viewstamped **O**peration **R**eplicator - Deterministic Simulation Testing inspired by FoundationDB:

- **The VOPR** runs 24/7 on 1,000 cores ("VOPR-1000")
- **Fault injection** - network drops, disk corruption, crashes, clock skew
- **1000x time acceleration** - 1 hour = 1 month, 1 day = 2 years
- **~2,000 years of simulated runtime per day**
- **Fully reproducible** - any bug replayed with just a seed number

Tests actual production code, not mocks. A crash is caught; silent corruption cannot hide.

Jepsen Report (June 2025)

Independent verification by Kyle Kingsbury (Aphyr):

- **Only 2 minor safety issues** - both fixed quickly
- **"Exceptional resilience to disk corruption"** - recovered from bitflips across replicas
- **Strong Serializability confirmed** under crashes, partitions, clock errors

Compare to other Jepsen results:

- Redis-Raft: 21 issues including data loss on failover
- Dgraph: data corruption even in healthy clusters
- MongoDB: lost majority-acknowledged writes
- Hazelcast: map updates silently lost

Benchmark Results

Benchmark	Contention	Result
vs PostgreSQL (single instance)	10%	**266x faster**
vs PostgreSQL cluster (16 nodes)	50%	**2,400x faster**
vs DuckDB (embedded, no network)	90%*	**229x faster**

*Models India's UPI where 85% of transactions flow through PhonePe and Google Pay

TigerBeetle's throughput remains **consistent regardless of contention**.

Traditional databases degrade exponentially as contention increases.

Why It’s Fast

- **Fixed schema, zero copy** - same 128-byte structure everywhere
- **io_uring** - Linux’s modern async I/O, minimal syscall overhead
- **Direct I/O** - bypasses OS page cache, deterministic behavior
- **Single-threaded** - counterintuitively faster for contended workloads
- **Batching at every level** - amortizes overhead

Client Libraries

Clients send and receive frames containing sequences of 128-byte operations.

Same binary protocol everywhere - official clients:

.NET	Go	Java
JavaScript	Python	Rust

No built-in auth - TigerBeetle is for trusted environments.

Or use `tigertunnel` : https://github.com/jedisct1/tigertunnel

Where TigerBeetle Fits

OLGP

General Purpose

PostgreSQL, MySQL

User profiles, config, metadata

OLTP

Transaction Processing

TigerBeetle

Transactions, balances

OLAP

Analytics

DuckDB, ClickHouse

Reports, analytics

TigerBeetle **complements** your existing database - it doesn't replace it.

TigerBeetle in Practice

Just One Executable

No configuration file needed.

Create storage:

```
tigerbeetle format --cluster=0 --replica=0 --replica-count=1 /tmp/tb/0
```

Run the server:

```
tigerbeetle start --addresses=127.0.0.1:2000 /tmp/tb/0
```

Production Setup (6 Replicas)

```
tigerbeetle start \
--cache-grid=4GiB \
--addresses=10.0.0.1:2000,10.0.0.2:2000,10.0.0.3:2000,10.0.0.4:2000,10.0.0.5:2000,10.0.0.6:2000 \
/tmp/tb/0
```

Connect via REPL:

```
tigerbeetle repl --cluster=0 --addresses=127.0.0.1:2000
```

Creating Accounts

```
create_accounts id=10 code=1 ledger=1
    flags=history|debits_must_not_exceed_credits;
```

- `code` : Chart of accounts code (u16) - describes account type (clearing, settlement, etc.)
- `ledger` : Partitions accounts by asset/currency

Account Flags

Flag	Description
`linked`	Links with next - all succeed or fail together
`debits_must_not_exceed_credits`	Asset account constraint
`credits_must_not_exceed_debits`	Liability account constraint
`history`	Enables `get_account_balances` queries
`imported`	For data migration scenarios
`closed`	Rejects further transfers

Creating Multiple Accounts

Create linked accounts in one batch:

```
create_accounts
    id=10 code=1 ledger=1 flags=history|debits_must_not_exceed_credits,
    id=11 code=1 ledger=1 flags=history|debits_must_not_exceed_credits|linked,
    id=12 code=1 ledger=1 flags=history|debits_must_not_exceed_credits;
```

Also create an "operator" account without balance constraints:

```
create_accounts id=100 code=1 ledger=1;
```

Creating Transfers

```
create_transfers id=1 code=1 ledger=1
    debit_account_id=10 credit_account_id=11 amount=100;
```

If debit account doesn’t have enough credits:

```
Failed to create transfer (0):
    tigerbeetle.CreateTransferResult.exceeds_credits.
```

Retry with same ID after failure:

```
Failed to create transfer (0):
    tigerbeetle.CreateTransferResult.id_already_failed.
```

Linked Transfers (Atomic Operations)

```
create_transfers
    id=7 code=1 ledger=1
    debit_account_id=100 credit_account_id=11
    amount=400 flags=linked,

    id=8 code=1 ledger=1
    debit_account_id=100 credit_account_id=11
    amount=500;
```

Both transfers succeed or fail together - atomic multi-transfer transactions!

Lookups

```
> lookup_accounts id=11
{
  "id": "11",
  "debits_pending": "0",
  "debits_posted": "0",
  "credits_pending": "0",
  "credits_posted": "1800",
  "ledger": "1",
  "code": "1",
  "flags": ["linked","debits_must_not_exceed_credits","history"],
  "timestamp": "1764591888414419002"
}
```

Query Transfer History

```
> get_account_transfers account_id=11;
{ "id": "2", "amount": "100", "flags": [], ... }
{ "id": "3", "amount": "100", "flags": [], ... }
{ "id": "4", "amount": "200", "flags": [], ... }
{ "id": "7", "amount": "400", "flags": ["linked"], ... }
{ "id": "8", "amount": "500", "flags": [], ... }
```

Every transfer that touched this account, with full details.

Balance History

```
> get_account_balances account_id=11;
{ "credits_posted": "100", "timestamp": "1764591942917580001" }
{ "credits_posted": "200", "timestamp": "1764591965603187001" }
{ "credits_posted": "400", "timestamp": "1764591972512163001" }
{ "credits_posted": "900", "timestamp": "1764591978265004002" }
{ "credits_posted": "1800", "timestamp": "1764592001779290002" }
```

Full audit trail - requires `history` flag on account.

Two-Phase Transfers

Reserve, Then Resolve

Phase 1: Reserve (Pending)

```
create_transfers
    id=10 code=1 ledger=1
    debit_account_id=11
    credit_account_id=12
    amount=500
    flags=pending
    timeout=60;
```

Phase 2: Resolve

Post (finalize):

```
create_transfers
    id=50 pending_id=10
    flags=post_pending_transfer;
```

Or void (cancel):

```
create_transfers
    id=50 pending_id=10
    flags=void_pending_transfer;
```

Transfer Flags

Flag	Description
`linked`	Links with next - all succeed or fail atomically
`pending`	Creates a two-phase pending transfer (requires timeout)
`post_pending_transfer`	Posts (finalizes) a pending transfer
`void_pending_transfer`	Voids (cancels) a pending transfer
`balancing_debit`	Auto-adjusts amount to debit account's limit
`balancing_credit`	Auto-adjusts amount to credit account's limit
`closing_debit`	Closes debit account (must combine with `pending`)

Pending Transfer State

```
> get_account_balances account_id=12;
{
  "debits_pending": "0",
  "debits_posted": "0",
  "credits_pending": "500",   // Reserved, not yet posted
  "credits_posted": "0",
  "timestamp": "..."
}
```

Pending amounts are reserved but not finalized until posted or voided.

Beyond Finance

Rate Limiting with Pending Transfers

Using pending transfers with timeouts for rolling windows:

```
// Create daily (86400s) and weekly (604800s) rate buckets
create_transfers
    id=1 debit_account_id=operator credit_account_id=user_daily
        ledger=daily_limit amount=100000,
    id=2 debit_account_id=operator credit_account_id=user_weekly
        ledger=weekly_limit amount=500000;

// For each API request: reserve from both buckets
create_transfers
    id=1001 debit_account_id=user_daily credit_account_id=operator
        amount=2000 timeout=86400 flags=pending|linked,
    id=1002 debit_account_id=user_weekly credit_account_id=operator
        amount=2000 timeout=604800 flags=pending;
```

Pending transfers auto-expire, giving rolling windows. Link both to fail atomically!

Multi-Ledger Atomic Transactions

Concert ticket purchase - payment AND inventory in one atomic operation:

```
create_transfers
    // Payment to seller
    id=1000 debit_account_id=buyer credit_account_id=seller
            amount=9000 ledger=currency flags=pending|linked,

    // Platform fee
    id=1001 debit_account_id=buyer credit_account_id=platform
            amount=1000 ledger=currency flags=pending|linked,

    // Transfer ticket
    id=1002 debit_account_id=seller_inventory credit_account_id=buyer
            amount=1 ledger=tickets flags=pending;
```

All three succeed or fail together!

LLM Proxy: Processing a Request

Atomic chain: rate limits + billing + provider payment in one batch:

```
create_transfers
    // Reserve daily quota (auto-expires in 24h)
    id=1001 debit_account_id=4242101 credit_account_id=2
        ledger=210 amount=2000 flags=pending|linked timeout=86400,

    // Reserve weekly quota (auto-expires in 7d)
    id=1002 debit_account_id=4242201 credit_account_id=3
        ledger=211 amount=2000 flags=pending|linked timeout=604800,

    // Pay the provider
    id=1003 debit_account_id=4242001 credit_account_id=9001
        ledger=100 amount=6000 flags=linked,

    // Platform takes its cut
    id=1004 debit_account_id=4242001 credit_account_id=4
        ledger=100 amount=2000;
```

All four succeed or fail together. Rate limits auto-reset via pending expiry!

Key Takeaways

Specialized, not general

Complements PostgreSQL for high-contention OLTP

Contention is the enemy

Traditional DBs degrade; TigerBeetle doesn't

Simple is powerful

128-byte transfers, batching, single-threaded

Beyond finance

Rate limiting, inventory, any counting workload

Resources

Website: tigerbeetle.com

Documentation: docs.tigerbeetle.com

GitHub: github.com/tigerbeetle/tigerbeetle

Simulator: sim.tigerbeetle.com

Questions?