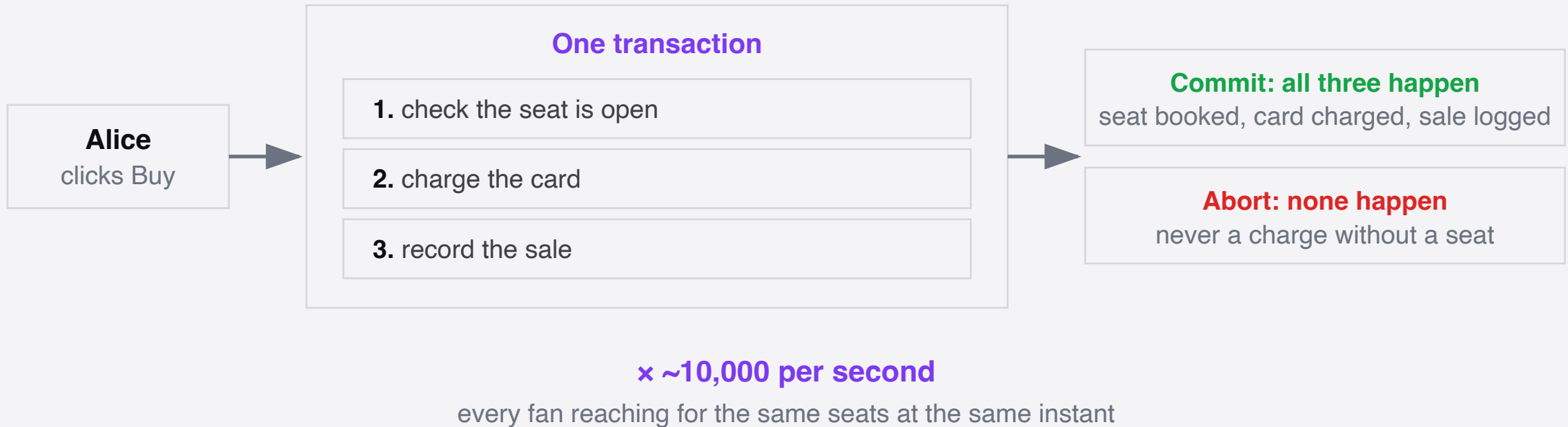


M 4

Transactions Motivation

A transaction groups related database operations into one all-or-nothing unit.

One Buy click is one transaction: three steps, all or none



One fan's Buy click is a single transaction: check the seat is open, charge the card, record the sale, bundled so either all three commit or none do, never a charge without a seat and never the same seat to two fans. Multiply by the on-sale crush, every fan racing for the same seats, and you have the problem this module solves.

Taylor Swift's Eras Tour on sale

52

show dates

3.5M

fans pre-registered

2M+

tickets sold
in a single day

3.5B

system requests
in a few hours

Ticketmaster reported 3.5 million registrations and 3.5 billion system requests, four times its prior traffic peak. Sales were delayed and fans encountered errors. The transaction guarantees are simple: no seat sold twice and no charge without a seat. Keeping those guarantees while staying responsive under load is the problem this module addresses.

Heavy demand tests availability and correctness

Reported demand

3.5M registrations

3.5B requests

4× the prior traffic peak

Availability suffered

Long waits and errors

Sales slowed or postponed

Fans lost carted tickets

Required guarantees

No seat sold twice

No charge without a seat

All or nothing

Ticketmaster reported 3.5 million registrations and 3.5 billion system requests, four times its prior traffic peak. Sales were delayed and fans encountered errors. The transaction guarantees are simple: no seat sold twice and no charge without a seat. Keeping those guarantees while staying responsive under load is the problem this module addresses.

Large workloads need deliberate engineering

Visa network

65,000+ transaction messages/sec capacity

Payment network

Amazon DynamoDB · Prime Day 2022

105.2M requests/sec · single-digit ms

Database requests

Ticketmaster · Eras Tour presale

4× its previous traffic peak

Availability disrupted

Visa reports capacity above 65,000 transaction messages per second. During Prime Day 2022, Amazon DynamoDB peaked at 105.2 million requests per second with single-digit millisecond responses. Ticketmaster reported four times its previous traffic peak. These are different workloads and metrics, but all require deliberate engineering for concurrency and availability.

More writes, the same correctness guarantees

Classic databases

1990s–2000s

Retail · Walmart

Illustrative write throughput

~10,000 writes/sec

Hardware

HDDs · a few CPU cores

Data structures

B-trees

Cloud services

2010s–2020s

Gmail · Roblox

Illustrative write throughput

Millions of writes/sec

Hardware

SSDs · many CPU cores

Data structures

B-trees + LSM trees

Agent workloads

2025 onward

Many autonomous clients

Illustrative write throughput

Tens of millions/sec

Hardware

NVMe · distributed compute

Data structures

Workload-specific indexes

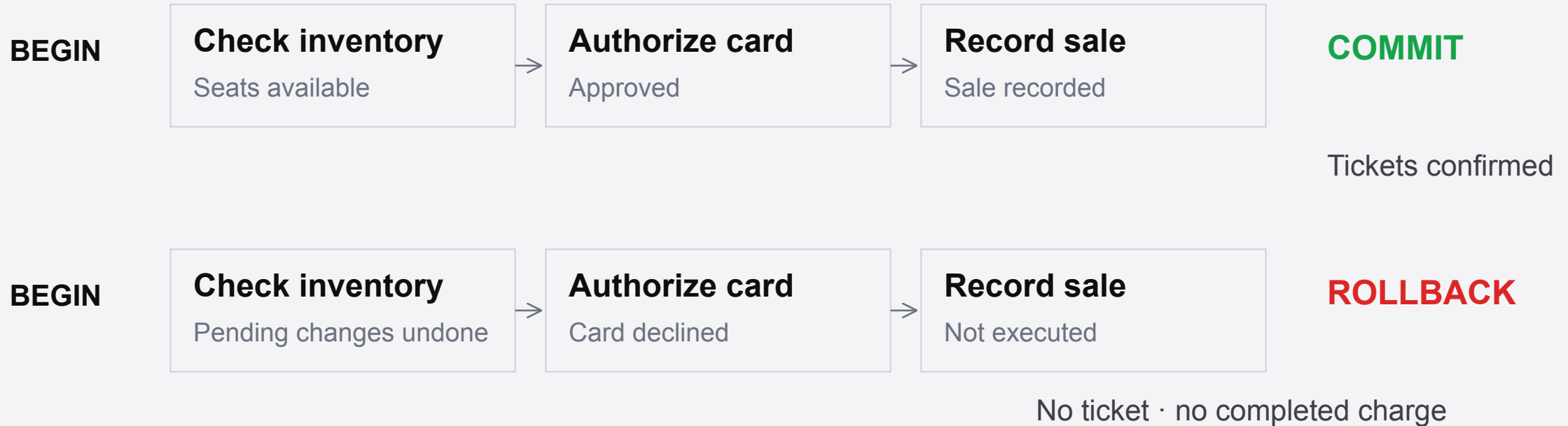
Classic retail systems such as Walmart, cloud services such as Gmail and Roblox, and agent workloads from 2025 onward illustrate increasing write demand: about 10,000 writes per second, millions, and a target of tens of millions. These are illustrative scale tiers, not measured rates for the named services. Hardware progresses from disks and a few cores to SSDs and many cores, then NVMe with distributed compute; every tier still needs correct transactions.

M 4

ACID Properties

ACID defines four guarantees for a transaction.

Alice's ticket purchase: commit or rollback



Alice's purchase checks inventory, requests card authorization, and records the sale. The database commits its pending changes together or rolls them back on failure. A declined authorization leaves no confirmed sale or completed charge. Coordination with the external payment network is application logic; a database rollback alone cannot reverse an external charge.

ACID: four guarantees, four examples

A Atomicity

All writes commit together, or none do.

Example

Card declines: no ticket, no charge, no change.

C Consistency

Every commit preserves the rules.

Example

Seats sold and revenue stay in sync.

I Isolation

Transactions hide their unfinished work.

Example

Alice locks the last seat; Bob waits.

D Durability

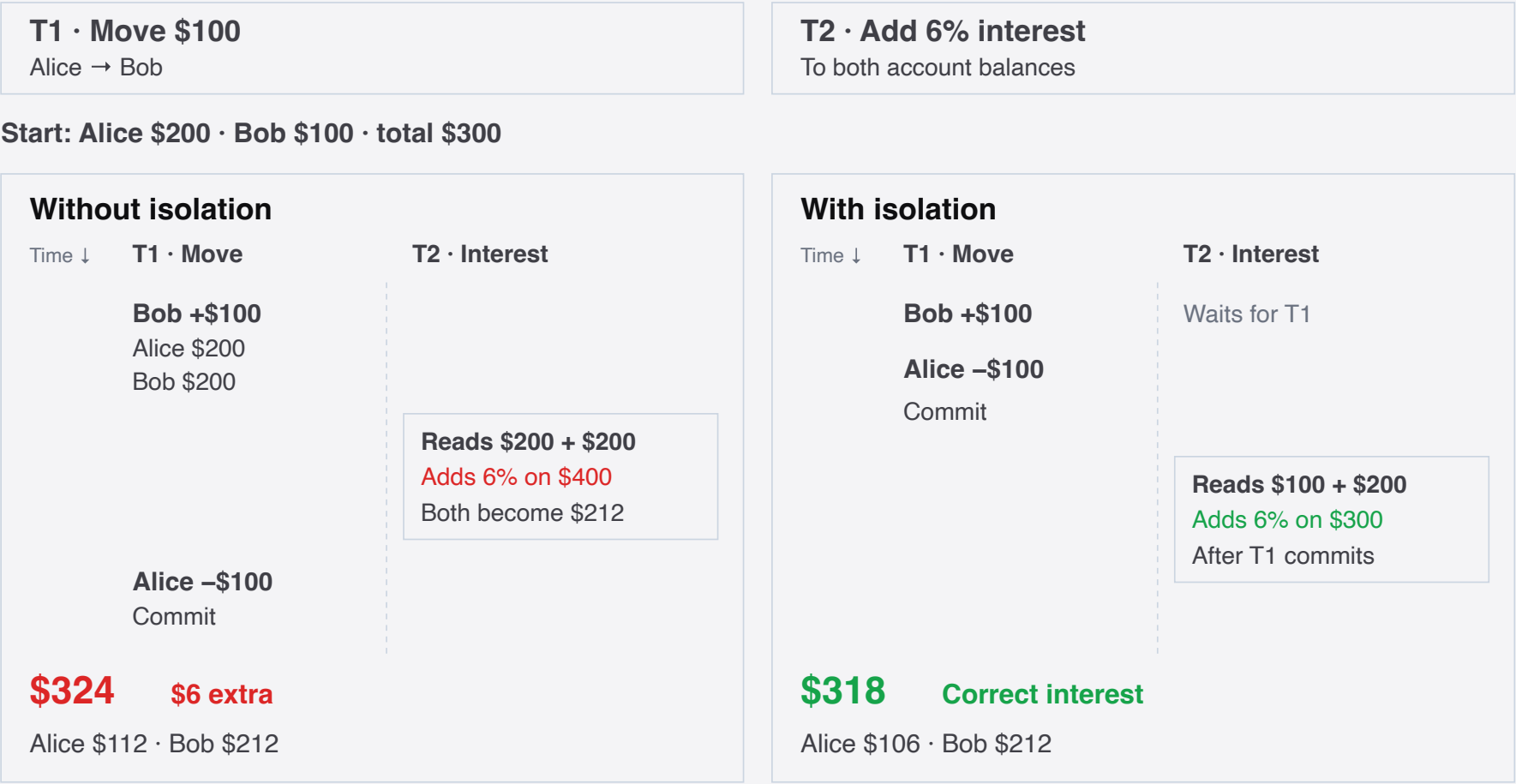
Committed changes survive a crash.

Example

Server crashes after confirmation.
The ticket remains recorded.

Each card separates a guarantee from its ticket-purchase example: a declined card rolls back cleanly (atomicity), seats and revenue move as one (consistency), Alice locks the last seat so Bob waits (isolation), and a confirmed ticket survives a crash (durability).

Interest must not see a half-finished transfer

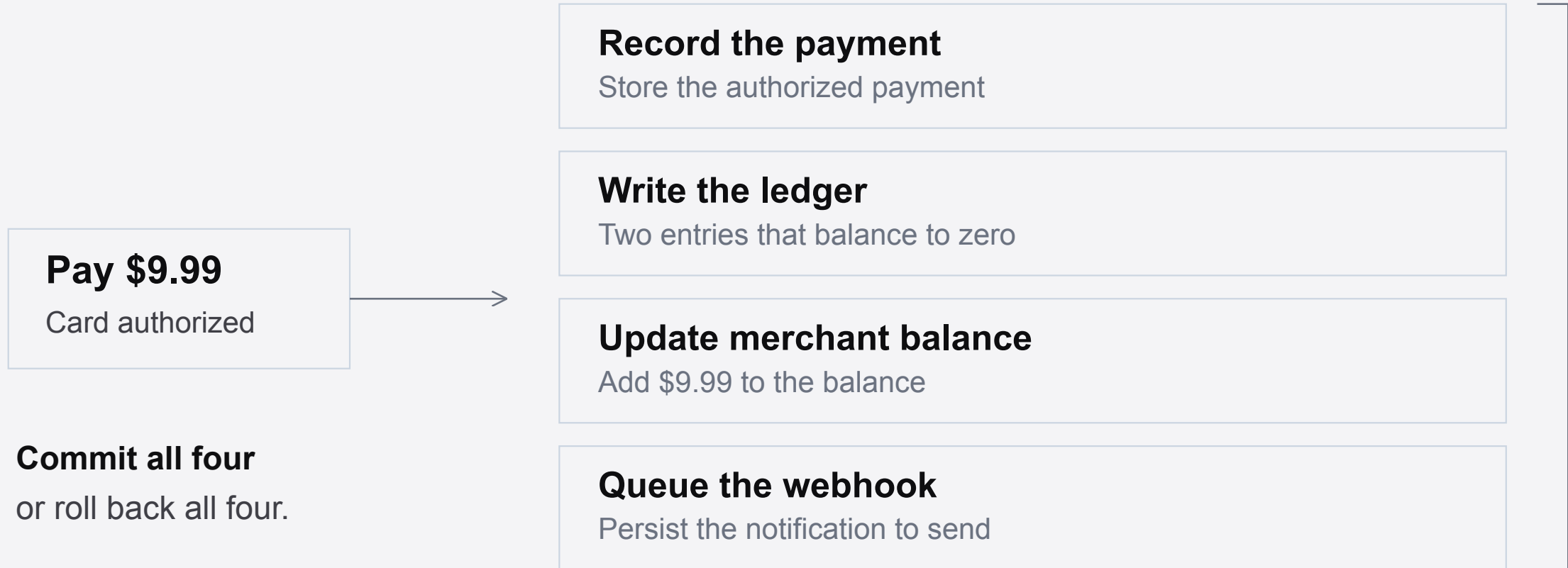


M 4

How ACID Protects a Stripe Payment

One payment updates several records that must agree.

One transaction commits four internal payment writes



After card authorization, the database records the payment, writes two balancing ledger rows, updates the merchant balance, and queues a webhook. One transaction commits those four internal writes together.

Four ACID properties prevent four payment failures

The partial internal state

Payment recorded. Webhook missing.
The database commits only half the work.

A **Atomicity** all of it, or none

The ledger that drifts

\$9.99 debited, \$9.98 credited.
The two rows stop summing to zero.

C **Consistency** the books always balance

The lost update

Two charges hit one balance at once.
Both read it, both write, one is lost.

I **Isolation** serial order, nothing lost

Lost to a crash

Receipt shown, then the server dies
five milliseconds later.

D **Durability** survives the crash

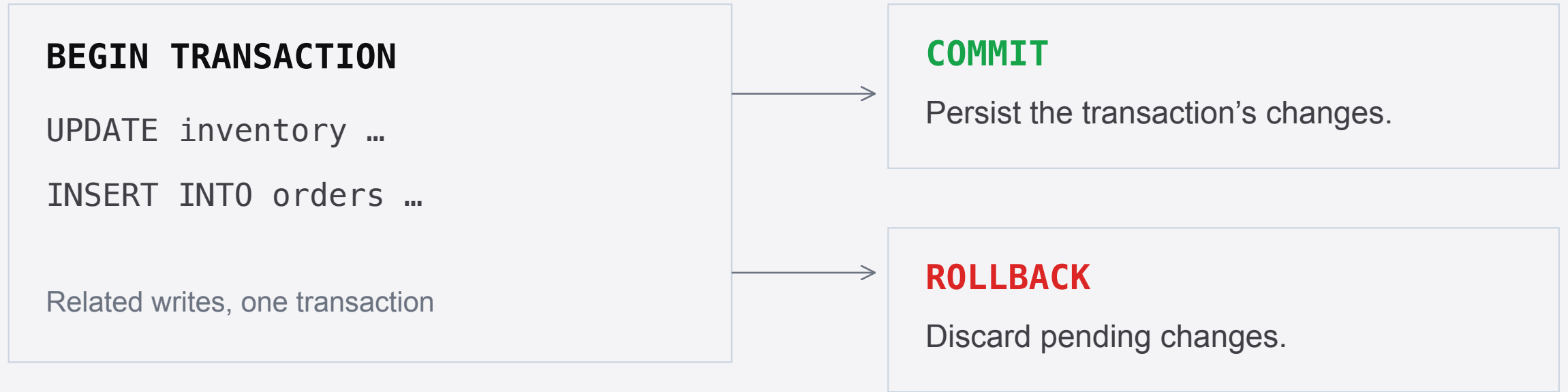
Four failures map to four guarantees: atomicity prevents partial internal writes, consistency keeps the ledger balanced, isolation prevents concurrent payments from losing a balance update, and durability preserves a confirmed payment after a crash.

M 4

Sample Code: Running Transactions

Use BEGIN and COMMIT to group statements into one transaction.

BEGIN, COMMIT, ROLLBACK



Autocommit

Outside the block, each statement is its own transaction.

The three keywords frame one block. `BEGIN` opens the transaction; your statements run inside as one logical operation; `COMMIT` makes every change since `BEGIN` permanent, or `ROLLBACK` undoes them all as if nothing happened. Rolling a transaction back is also called aborting it, the term the rest of this module uses (in Postgres, `ABORT` is the same command). Outside a block, each statement autocommits as its own one-statement transaction.

Buy tickets

```
BEGIN TRANSACTION;

SELECT available_tickets
FROM Concerts
WHERE concert_id = 'taylor_swift_123';

UPDATE Concerts
SET available_tickets = available_tickets - 2
WHERE concert_id = 'taylor_swift_123';

INSERT INTO Orders (user_id, concert_id, num_tickets, total_amount, status)
VALUES ('user_789', 'taylor_swift_123', 2, 150.00, 'confirmed');

COMMIT;
```

Transfer money

```
-- Schema rule: CHECK (balance >= 0).  
-- The database enforces this on every write.
```

```
BEGIN TRANSACTION;
```

```
UPDATE Accounts  
SET balance = balance - 100.00  
WHERE account_id = 'alice_account';
```

```
UPDATE Accounts  
SET balance = balance + 100.00  
WHERE account_id = 'bob_account';
```

```
INSERT INTO Transfers (from_account, to_account, amount, type)  
VALUES ('alice_account', 'bob_account', 100.00, 'transfer');
```

```
COMMIT;
```

Reserve inventory

```
BEGIN TRANSACTION;
```

```
SELECT quantity FROM Products WHERE product_id = 'labubu';
```

```
UPDATE Products SET quantity = quantity - 1 WHERE product_id = 'labubu';
```

```
INSERT INTO Orders (customer_id, product_id, quantity, status)  
VALUES ('customer_alice', 'labubu', 1, 'confirmed');
```

```
COMMIT;
```

Record a like

```
BEGIN TRANSACTION;
```

```
-- The Likes table has a UNIQUE (user_id, post_id) constraint; a duplicate  
-- INSERT fails and the whole transaction rolls back.
```

```
INSERT INTO Likes (user_id, post_id, timestamp)  
VALUES ('user_12345', 'viral_post_456', NOW());
```

```
UPDATE Posts
```

```
SET like_count = like_count + 1,  
    last_activity = NOW()  
WHERE post_id = 'viral_post_456';
```

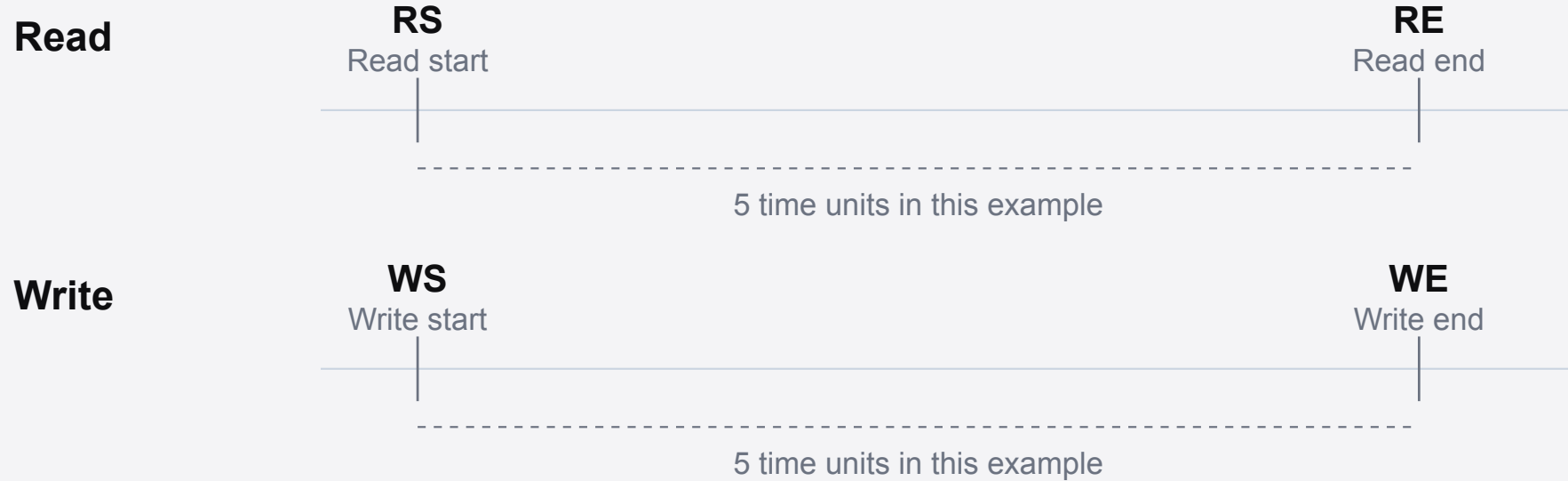
```
COMMIT;
```

M 4

Naive Concurrency

Concurrency control keeps shared data correct while transactions run together.

I/O lag: reads and writes take time

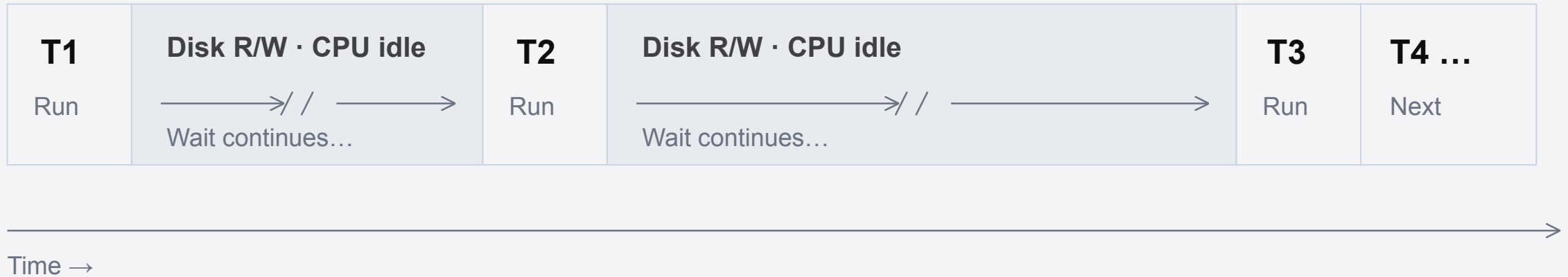


$$\text{I/O lag} = \text{end time} - \text{start time}$$

I/O lag is the time from read start (RS) to read end (RE), or write start (WS) to write end (WE). Here it takes 5 time units. In practice, I/O can take millions of CPU cycles.

Naive Way 1: run one at a time

Server



T3 and T4 wait behind T2, even while the CPU is idle.

The disk handles reads and writes while serial execution leaves the CPU idle. The broken arrows compress the waiting spans; they do not represent measured durations. T3 and T4 cannot run while T2 waits, even when their work touches unrelated rows.

Naive Way 2: run without coordination

Lost update

write – write conflict

play_count = 100

T1

read → 100

write 101

T2

read → 100

write 101

lands on 101, not 102

one play silently vanishes

Dirty read

write – read conflict

balance = 100

T1

write 150

ABORT

T2

read → 150

acts on 150

T2 used 150

a value that never committed

Stale read

read – write conflict

price = 20

T1

read → 20

read → 30

T2

write 30

commit

T1 saw 20, then 30

two reads disagree in one txn

Uncontrolled concurrency corrupts one row three different ways. In a lost update, two transactions both read 100 and both write 101, so one play disappears. In a dirty read, one transaction reads another's uncommitted write and the writer then aborts, so the reader acted on a value that never existed. In a stale read, a transaction reads the same row twice and a committed write in between makes its two reads disagree.

How should transactions share the database?

Serial

Run one transaction at a time.
Disk waits leave the CPU idle.

Uncoordinated

Run transactions without checks.
Conflicting writes can lose updates.

Concurrency control

Overlap independent work.
Order conflicting operations safely.

Concurrency control overlaps independent work while ordering conflicting actions safely.

M 4

A Quick Graph Refresher

A directed graph captures which operations must come before others.

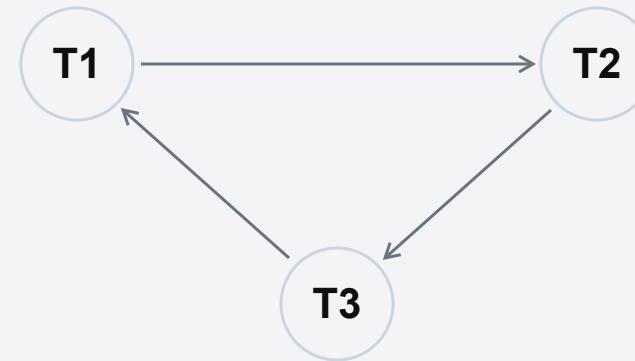
Acyclic graphs have no directed cycle

Acyclic (DAG)



An order respects every edge.

Cyclic

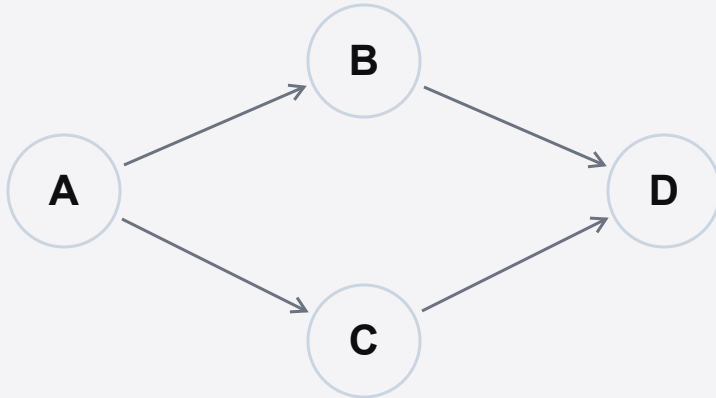


$T1 \rightarrow T2 \rightarrow T3 \rightarrow T1$

Left: no directed cycle, so an order respecting every edge exists. Right: the cycle prevents such an order. A transaction schedule is conflict-serializable exactly when its conflict graph has no cycle.

A topological order respects every edge

Dependencies



Two valid orders

$A \rightarrow B \rightarrow C \rightarrow D$

$A \rightarrow C \rightarrow B \rightarrow D$

B and C can appear in either order.

Both $A \rightarrow B \rightarrow C \rightarrow D$ and $A \rightarrow C \rightarrow B \rightarrow D$ respect every edge. B and C have no ordering constraint between them. In a conflict graph, a topological order gives an equivalent serial transaction order.