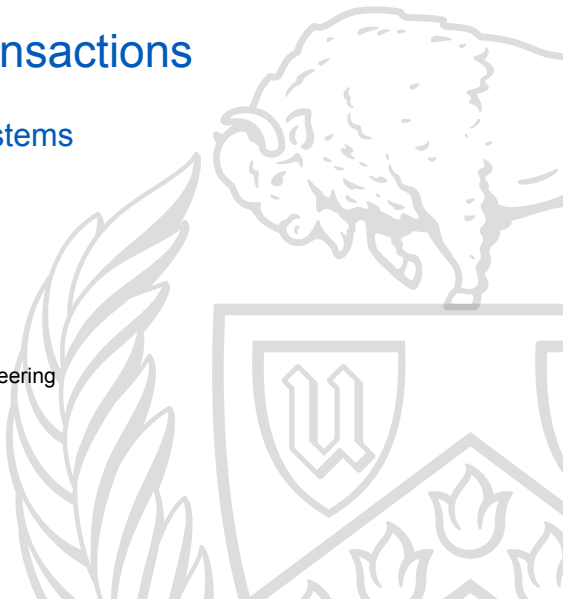


Consistency and Transactions

CSE 486/586: Distributed Systems

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo



Consistency

Many applications have **consistency** requirements.

Sometimes we think of these as **invariants**:

- The number of items in inventory cannot fall below zero
- The forward and reverse pointers in a doubly-linked list will be reflexive

Sometimes they are **conditional**:

- If a debit fails, the corresponding credit must also fail
- A file transfer must be verified complete before the source is removed

Distributed Consistency

In a distributed system, **consistency may require consensus**.

- What if inventory is spread through warehouses across the country?
- What if the debiting account and crediting account are at different institutions?

This introduces FLP to the mix!

Failure detection and **timeouts** are often used.

(We won't look at this just yet.)

Transactions

Consistency may be violated **during a computation**.

For example, that bank transfer:

1. A quantity is debited from one account
2. The same quantity is deposited to another account

In between, **the accounts are inconsistent**: money is lost!

These actions together form a **transaction** [2].

Consistency is maintained **before** and **after** the transaction.

Atomic Transactions

With concurrency, transactions require **atomicity**.

Recall:

- If a debit fails, the corresponding credit must also fail

What happens if:

- The debit succeeds but the credit fails?
- The debit fails but the credit succeeds?

A transaction must:

- Never expose **partially-complete results**
- Either **fully succeed** or **fail without effect**

Schedules

Each transaction is made up of individual **actions**.

A **schedule** is some sequence of those actions.

A **consistent schedule** is such a schedule that ensures that **each transaction** sees a **consistent state**.

Consistent schedules trivially **maintain consistency**.

Administrivia

Course evaluations are available!

Please fill out your evaluations.

I read every evaluation, and I make changes to courses.

Tell me:

- What worked for you
- What didn't work for you

Consistency

Transaction consistency depends on **correct transactions**.

We will **assume** that transactions are correct.

This allows us to state that:

If the system is consistent **before a transaction begins**, it is consistent **after the transaction completes**.

It is our job to **schedule transactions** to preserve this.

Atomicity

T1:

savings ← savings - \$200

checking ← checking + \$200

What if T1 fails between lines 1 and 2?

Transactions normally have **three operations**:

- begin
- commit
- abort

A committed transaction **is completely successful**.

An aborted transaction **never happens**.

Example Transactions

Suppose we have the consistency constraint $A = B$.

T1:

$$A \leftarrow A + 100$$

$$B \leftarrow B + 100$$

T2:

$$A \leftarrow A \times 2$$

$$B \leftarrow B \times 2$$

Is it OK to:

- Run T1, then T2?
- Run T2, then T1?
- Run line 1 of T1, then T2, then line 2 of T1?

Example Transactions

Suppose that T1 and T2 operate on accounts:

T1:

savings \leftarrow savings - \$100

checking \leftarrow checking + \$100

T2:

total \leftarrow savings

total \leftarrow total + checking

What if T2 runs between lines 1 and 2 of T1?

Does this represent the actual total of the accounts?

Concurrency

Why have concurrency at all?

We can just run transactions **one at a time!**

Consider: ISIS, Raft.

This is **serial execution** and it **ensures consistency**.

It is also **inefficient**:

- Communication is **orders of magnitude** slower than simple computation!
- Transactions may require **arbitrarily complex** computation

Transaction Independence

Some transactions can **freely run concurrently**.

For example: two transactions on **disjoint** state.

T1:

$A \leftarrow A + 100$

T2:

$B \leftarrow B - 200$

$C \leftarrow C + 200$

There is **no ordering** of these operations that is incorrect!

All orderings are **equivalent** to running T1, then T2.

Transaction Conflicts

In general, two transactions **can have problematic ordering** if:

- Both transactions use some state S
- **At least one** transaction **changes** S

We divide these into three conflict types:

- **read-write**: T1 reads **changing values** for some state
- **write-read**: T1 reads a value which is **not committed**
- **write-write**: T1's write is **overwritten**

Note that **there is no read-read conflict**.

Conflicts between operations can be modeled as **graphs** [1].

Read-Write Conflict

T1:

read A
compute
read A
write something
commit

T2:

write A
commit

It is a **read-write** conflict if T2 executes during “compute” in T1.

Write-Read Conflict

T1:

read A
read B
write something
commit

T2:

write A
compute
write B
commit

It is a **write-read** conflict if T1 executes during “compute” in T2.

Write-Write Conflict

T1:

read A
write A
compute
write B
commit

T2:

read B
write A
commit

It is a **write-write** conflict if T2 executes during “compute” in T2.

Serializability

We wish to **interleave transactions** to maintain efficiency.

Many more transactions per **unit time** can be processed this way.

To maintain consistency, we preserve **serializability**.

Two transactions T1 and T2 are **serializable** if:

- to an **external observer**,
- it **appears as if** one **happened before** the other

E.g., T1 happened before T2.

Actions and Transactions

Serializability is determined from the **actions** in the transaction.

A schedule of actions is **serializable** if it is equivalent to some **serial execution** of the same transactions.

Formally:

Suppose that S is a schedule, and S_{SE} is its serial equivalent. For every **pair of conflicting actions** a_1, a_2 in S , if a_1 happens before a_2 in S then a_1 happens before a_2 in S_{SE} .

Example

T1:

read A
write B
commit

T2:

read B
write C
commit

read A
read B
write C
write B
commit
commit

$T2 \rightarrow T1$

Why not $T1 \rightarrow T2$?

Aborted Transactions

If transactions **can abort**, then there can be **cascading aborts** [3].

Cascading aborts are where:

- Some transaction T1 observes the output of T2
- T2 aborts instead of committing
- T1 must now abort to preserve **serializability**

Note that this **cannot happen** with serial execution:

T2 either **already committed** or **already aborted** before T1 began.

Two-Phase Locking

Transactions maintain serializability if they run in **two phases** [2]:

- **Growing Phase**: First, a transaction **acquires locks**
- **Shrinking Phase**: Second, a transaction **releases locks**

Not all locks must be acquired/released at the same time.

Once **any lock** is released, **no lock** can be acquired.

We call this **two-phase locking** (2PL).

Serializability of 2PL

This preserves serializability because:

- While T1 locks a datum, T2 cannot observe it
- Once T1 unlocks **any** datum, it will not modify **any observable** datum

Therefore, for **each datum**, either:

- T1 occurs before T2, or
- T2 occurs before T1

Non-serializability would **imply deadlock**. (We know how to avoid that!)

Intuition

There is a point in each transaction where:

- It has acquired all of its locks
- It has **not yet released** any lock

We call this the **lock point**.

Effectively, if T1's lock point is before T2's lock point:

- T1 **released every shared datum** before T2 locked it
- T1 is serializable before T2

2PL and Aborts

Two-phase locking does **not prevent** cascading aborts!

However, there is a modification that does: **Strict 2PL**.

In strict two-phase locking, **all locks are released at once**.

In this way, **no transaction output is visible** until it commits.

If no other transaction views its changes, **no aborts can cascade**.

Summary

- Transactions are **multiple actions** grouped together into an **atomic entity**.
- The actions in transactions can be **interleaved**.
- Some interleavings are **inconsistent**.
- Consistent interleavings are **serializable**.
- **Two-phase locking** preserves serializability.

References I

Optional Readings

- [1] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. “Formal Aspects of Serializability in Database Concurrency Control”. In: *IEEE Transactions on Software Engineering SE-5.3* (May 1979), pp. 203–216. DOI: 10.1109/TSE.1979.234182. URL: https://search.lib.buffalo.edu/permalink/01SUNY_BUF/12pkqkt/cdi_ieee_primary_1702620.

References II

- [2] Kapali. P. Eswaran et al. “The Notions of Consistency and Predicate Locks in a Database System”. In: *Communications of the ACM* 19.11 (Nov. 1976). Ed. by Howard L. Morgan, pp. 624–633. DOI: 10.1145/360363.360369. URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.386.9726&rep=rep1&type=pdf>.
- [3] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Fourth Edition. Springer, 2020. ISBN: 978-3-030-26252-5.

Copyright 2021 Ethan Blanton, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see <https://www.cse.buffalo.edu/~eblanton/>.