Concurrency Control

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: Computing Systems and Concurrency
Lecture 17

Marco Canini

Credits: Michael Freedman and Kyle Jamieson developed much of the original material.
Contents adapted from Wyatt Lloyd.

Let’s Scale Strong Consistency!

1. **Transactions and Atomic Commit review**

2. Serializability
   - Strict serializability

3. Concurrency Control:
   - Two-phase locking (2PL)
   - Optimistic concurrency control (OCC)

The transaction

- *Definition:* A unit of work:
  - May consist of **multiple** data accesses or updates
  - Must **commit** or **abort** as a **single atomic unit**

- Transactions can either **commit,** or **abort**
  - When **commit,** all updates performed on data are made permanent, visible to other transactions
  - When **abort,** data restored to a state such that the aborting transaction never executed

Transaction examples

- Bank account transfer
  - A -= $100
  - B += $100

- Maintaining symmetric relationships
  - A FriendOf B
  - B FriendOf A

Defining properties of transactions

- **Atomicity:** Either **all** constituent operations of the transaction complete successfully, or **none** do
- **Consistency:** Each transaction in isolation preserves a set of **integrity constraints** on the data
- **Isolation:** Transactions' behavior not impacted by presence of **other concurrent transactions**
- **Durability:** The transaction's **effects survive failure** of volatile (memory) or non-volatile (disk) storage

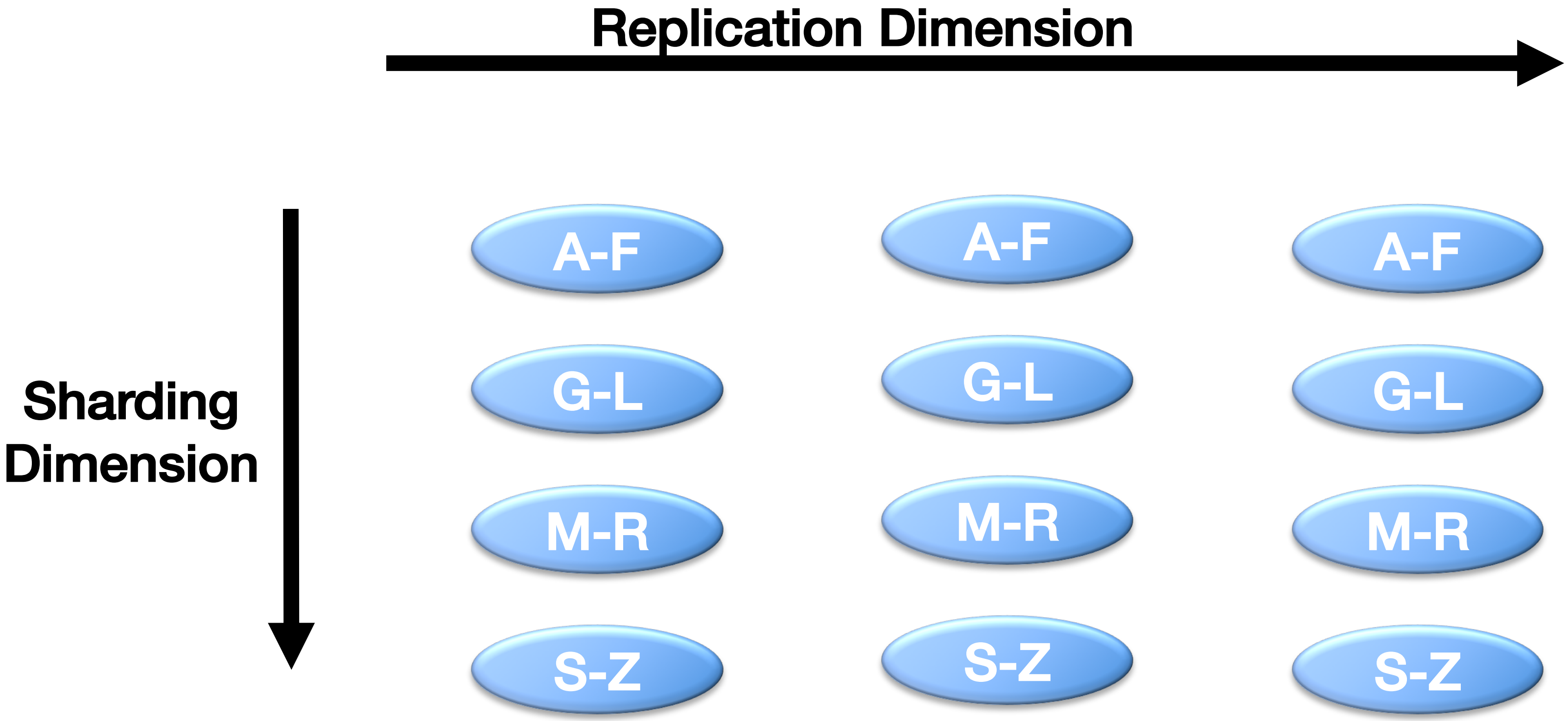
Atomic Commit

- Atomic: All or nothing
- Either all participants do something (commit) or no participant does anything (abort)
- Common use: commit a transaction that updates data on different shards

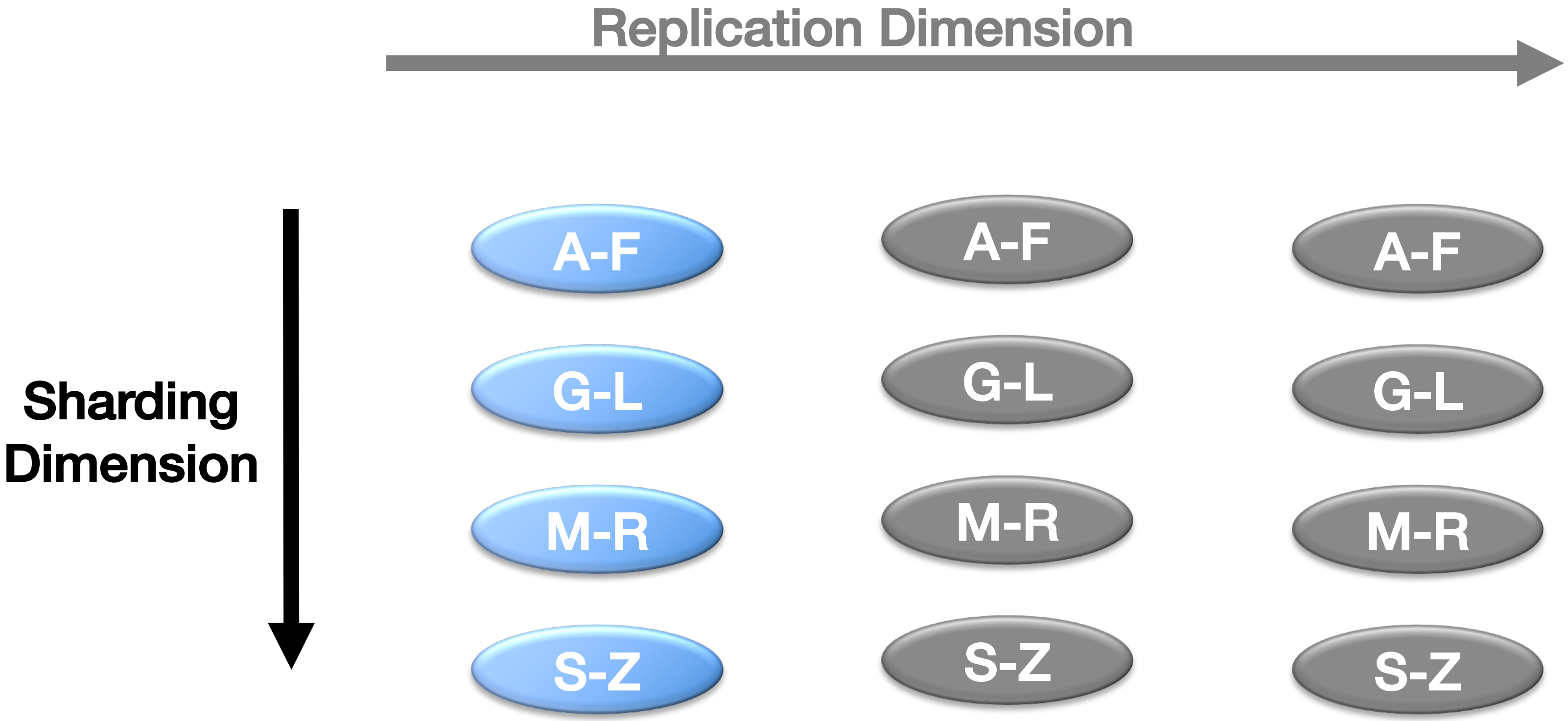
Relationship with replication

- Replication (e.g., RAFT) is about doing the same thing multiple places to provide fault tolerance
- Sharding is about doing different things multiple places for scalability
- Atomic commit is about doing different things in different places together

Relationship with replication



Focus on sharding for today



Atomic Commit

- Atomic: All or nothing
- Either all participants do something (commit) or no participant does anything (abort)
- Atomic commit is accomplished with the Two-phase commit protocol (2PC)

Let’s Scale Strong Consistency!

1. Transactions and Atomic Commit review

2. **Serializability**
    - **Strict serializability**

3. Concurrency Control:
    - Two-phase locking (2PL)
    - Optimistic concurrency control (OCC)

Two concurrent transactions

```
transaction sum(A, B):
begin_tx
a ← read(A)
b ← read(B)
print a + b
commit_tx
```

```
transaction transfer(A, B):
begin_tx
a ← read(A)
if a < 10 then abort_tx
else    write(A, a−10)
        b ← read(B)
        write(B, b+10)
        commit_tx
```

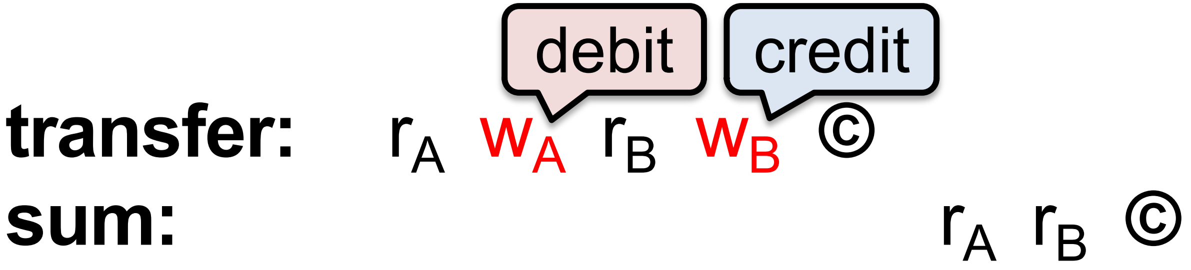
Isolation between transactions

- **Isolation:** **sum** appears to happen either completely before or completely after **transfer**
  - i.e., it appears that all operations of a transaction happened together
  - sometimes called *before-after atomicity*

- *Schedule* for transactions is an ordering of the operations performed by those transactions

Problem for concurrent execution: Inconsistent retrieval

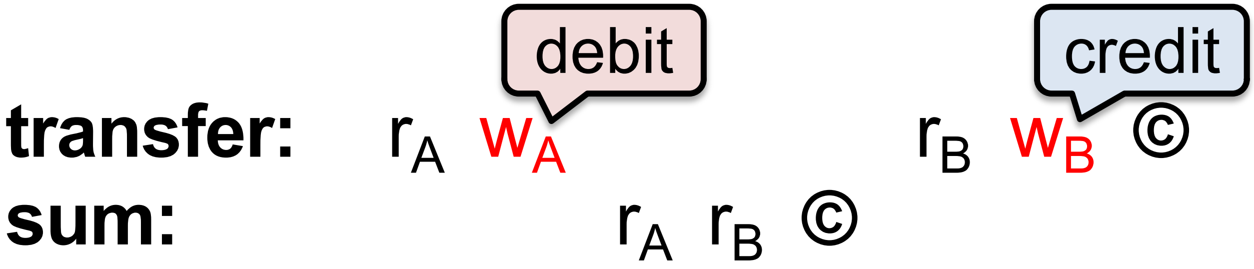
- **Serial execution** of transactions—transfer then sum:



transfer: r_A w_A r_B w_B ©

sum: r_A r_B ©

- Concurrent execution resulting in ***inconsistent retrieval,*** result differing from any serial execution:



transfer: r_A w_A r_B w_B ©

sum: r_A r_B ©

Time →

© = commit

Isolation between transactions

- **Isolation: sum** appears to happen either completely before or completely after **transfer**
  - i.e., it appears that all operations of a transaction happened together
  - sometimes called *before-after atomicity*

- Given a schedule of operations:
  - *Is that schedule in some way "equivalent" to a serial execution of transactions?*

Equivalence of schedules

- Two **operations** from **different transactions** are ***conflicting*** if:

1. They **read** and **write** to the **same data item**
2. The **write** and **write** to the **same data item**

- Two **schedules** are ***equivalent*** if:

1. They contain the same transactions and operations
2. They **order** all **conflicting** operations of non-aborting transactions in the **same way**

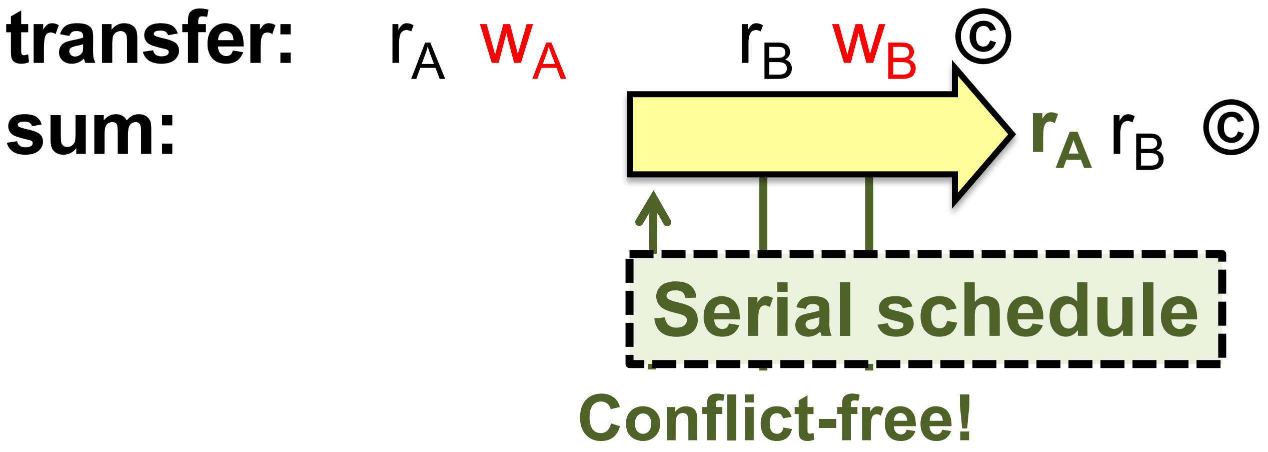
Serializability

- Ideal isolation semantics: *serializability*
- A schedule is ***serializable*** if it is equivalent to some serial schedule
  - *i.e.*, **non-conflicting** operations can be **reordered** to get a **serial** schedule

A serializable schedule

- Ideal isolation semantics: *serializability*

- A schedule is ***serializable*** if it is equivalent to some serial schedule
  - *i.e.*, **non-conflicting** operations can be **reordered** to get a **serial** schedule



transfer: r_A w_A r_B w_B ©

sum: r_A r_B ©

Serial schedule

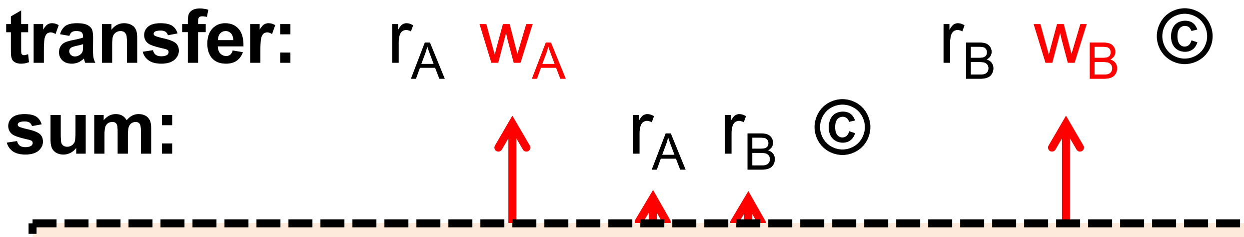
Conflict-free!

Time →

© = commit

A non-serializable schedule

- Ideal isolation semantics: *serializability*
- A schedule is ***serializable*** if it is equivalent to some serial schedule
  - *i.e.*, **non-conflicting** operations can be **reordered** to get a **serial** schedule



transfer:	r_A	w_A				r_B	w_B	©
sum:			r_A	r_B	©			

But in a **serial schedule**, sum's reads either **both before** w_A or **both after** w_B

Time →

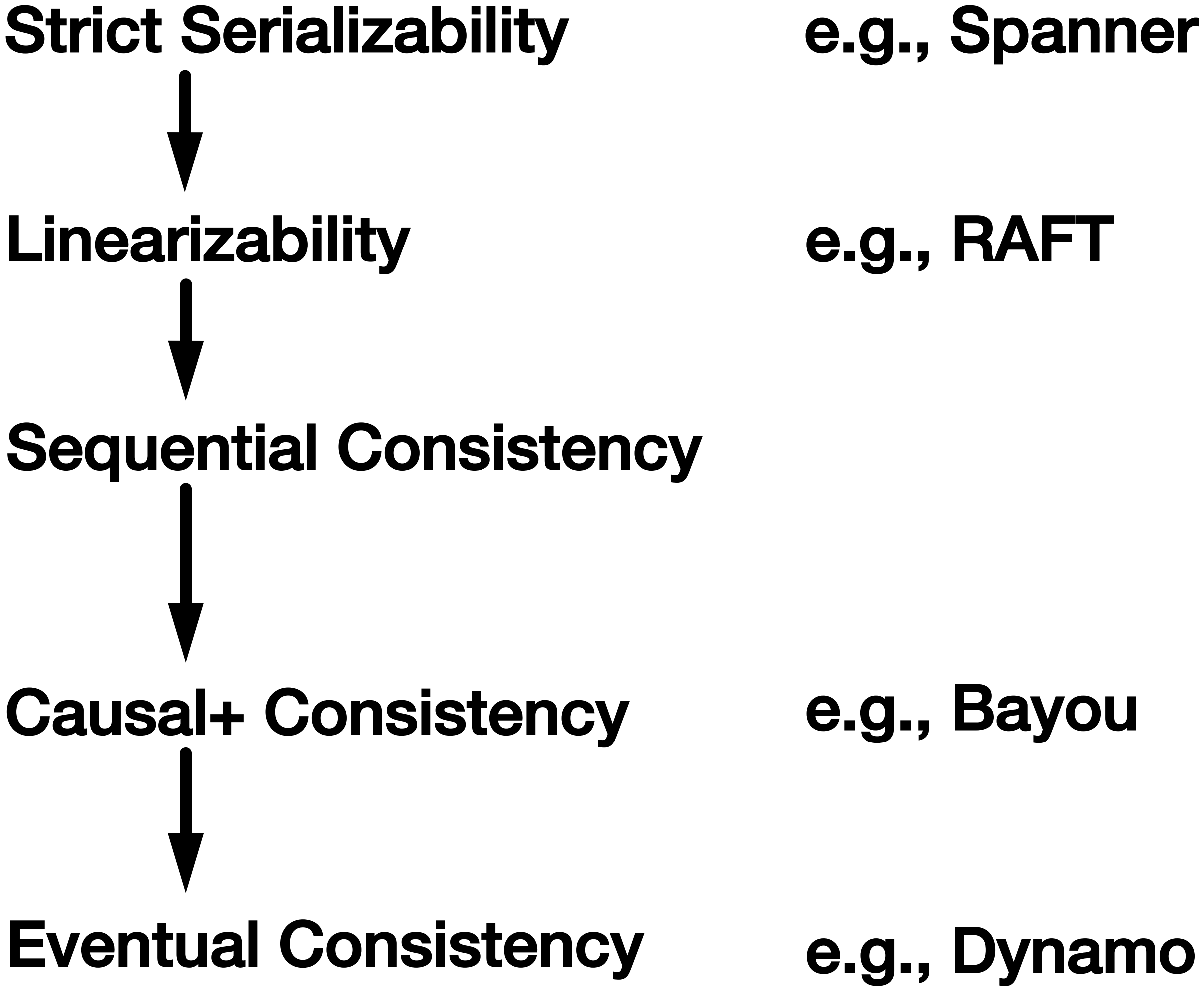
© = commit

Serializability versus linearizability

- **Linearizability:** a guarantee about **single** operations on **single** objects
  - Once write completes, all later reads (by wall clock) should reflect that write
- **Serializability** is a guarantee about **transactions** over **one or more** objects
  - Doesn't impose real-time constraints

- ***Strict serializability*** **= Serializability + real-time ordering**
  - Intuitively Serializability + Linearizability
  - Transaction behavior equivalent to some serial execution
    - **And** that serial execution **agrees with real-time**

Consistency Hierarchy



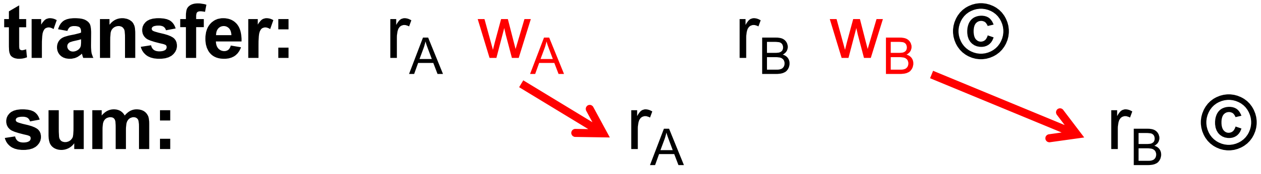
Strict Serializability	e.g., Spanner
↓	
Linearizability	e.g., RAFT
↓	
Sequential Consistency	
↓	
Causal+ Consistency	e.g., Bayou
↓	
Eventual Consistency	e.g., Dynamo

Testing for serializability

- Each node ***t*** in the ***precedence graph*** represents a transaction ***t***
  - Edge from ***s*** to ***t*** if some action of ***s*** **precedes and conflicts with** some action of ***t***

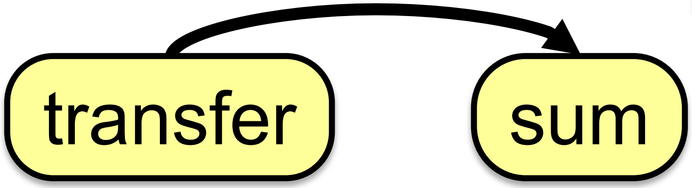
Serializable schedule, acyclic graph

- Each node ***t*** in the ***precedence graph*** represents a transaction ***t***
  - Edge from ***s*** to ***t*** if some action of ***s*** **precedes and conflicts with** some action of ***t***



transfer: r_A w_A r_B w_B ©

sum: r_A r_B ©



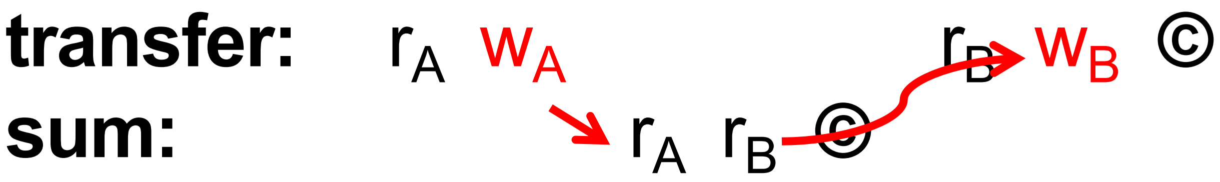
Serializable

Time →

© = commit

Non-serializable schedule, cyclic graph

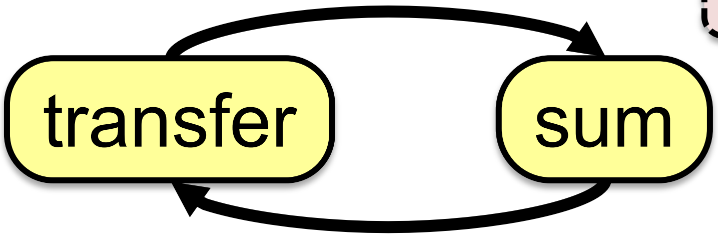
- Each node *t* in the ***precedence graph*** represents a transaction ***t***
  - Edge from ***s*** to ***t*** if some action of ***s*** **precedes and conflicts with** some action of ***t***



transfer: r_A w_A r_B w_B ©

sum: r_A r_B ©

Non-serializable



transfer

sum

Time →

© = commit

Testing for serializability

- Each node ***t*** in the ***precedence graph*** represents a transaction ***t***
  - Edge from ***s*** to ***t*** if some action of ***s*** **precedes and conflicts with** some action of ***t***

In general, a schedule is **serializable** if and only if its **precedence graph** is **acyclic**

Let’s Scale Strong Consistency!

1. Transactions and Atomic Commit review

2. Serializability
   – Strict serializability

3. **Concurrency Control:**
   – **Two-phase locking (2PL)**
   – **Optimistic concurrency control (OCC)**

Concurrency Control

- Concurrent execution can violate serializability
- We need to **control** that concurrent execution so we do things a single machine executing transactions one at a time would
  - Concurrency control

Concurrency Control Strawman #1

- **Big Global Lock**
  - Acquire the lock when transaction starts
  - Release the lock when transaction ends

- Provides strict serializability
  - Just like executing transaction one by one because we are doing exactly that

- No concurrency at all
  - Terrible for performance: one transaction at a time

Locking

- Locks maintained on each shard
  - Transaction requests lock **for a data item**
  - Shard **grants** or **denies** lock

- **Lock types**
  - ***Shared:*** Need to have before read object
  - ***Exclusive:*** Need to have before write object

	Shared (S)	**Exclusive (X)**
Shared (S)	Yes	No
Exclusive (X)	No	No

Concurrency Control Strawman #2

- Grab locks **independently**, for each data item (*e.g.*, bank accounts A and B)

transfer: $◢_A$ r_A w_A $◣_A$ $◢_B$ r_B w_B $◣_B$ ©

sum: $◿_A$ r_A $◺_A$ $◿_B$ r_B $◺_B$ ©

Permits this **non-serializable** interleaving

Time →

© = commit

◢ / ◿ = **eXclusive- / Shared-lock;** ◣ / ◺ = **X- / S-unlock**

Two-phase locking (2PL)

- **2PL rule:** Once a transaction has **released** a lock it is **not allowed to obtain** any other locks
  - **Growing phase** when transaction acquires locks
  - **Shrinking phase** when transaction releases locks
- In practice:
  - Growing phase is the entire transaction
  - Shrinking phase is during commit

2PL provides strict serializability

- **2PL rule:** Once a transaction has **released** a lock it is **not allowed to obtain** any other locks

transfer: ◢$_A$ r$_A$ w$_A$ ◣$_A$ 🚫$_B$ r$_B$ w$_B$ ◣$_B$ ©

sum: ◿$_A$ r$_A$ ◺🚫$_A$ ◿$_B$ r$_B$ ◺$_B$ ©

2PL **precludes** this **non-serializable** interleaving

Time →

© = commit

◢ / ◿ = **X- / S-lock;** ◣ / ◺ = **X- / S-unlock**

2PL and transaction concurrency

- **2PL rule:** Once a transaction has **released** a lock it is **not allowed to obtain** any other locks

transfer: ⊿$_A$ r_A ◢$_A$ w_A ⊿$_B$ r_B ◢$_B$ w_B ✱©

sum: ⊿$_A$ r_A ⊿$_B$ r_B ✱©

2PL **permits** this **serializable, interleaved** schedule

Time →

© = commit

◢ / ⊿ = **X- / S-lock;** ◣ / ◺ = **X- / S-unlock;** ✱ **= release all locks**

2PL doesn't exploit all opportunities for concurrency

- **2PL rule:** Once a transaction has **released** a lock it is **not allowed to obtain** any other locks

transfer:	r_A	w_A		r_B	w_B	©		
sum:			r_A				r_B	©

2PL **precludes** this **serializable, interleaved** schedule

Time →

© = commit

(locking not shown)

Issues with 2PL

- What do we do if a lock is unavailable?
  - Give up immediately?
  - Wait forever?

- Waiting for a lock can result in **deadlock**
  - Transfer has A locked, waiting on B
  - Sum has B locked, waiting on A

- Many ways to detect and deal with deadlocks
  - e.g., centrally detect deadlock cycles and **abort involved transactions**

Lets Scale Strong Consistency!

1. Atomic Commit
   - Two-phase commit (2PC)

2. Serializability
   - Strict serializability

3. Concurrency Control:
   - Two-phase locking (2PL)
   - Optimistic concurrency control (OCC)

2PL is pessimistic

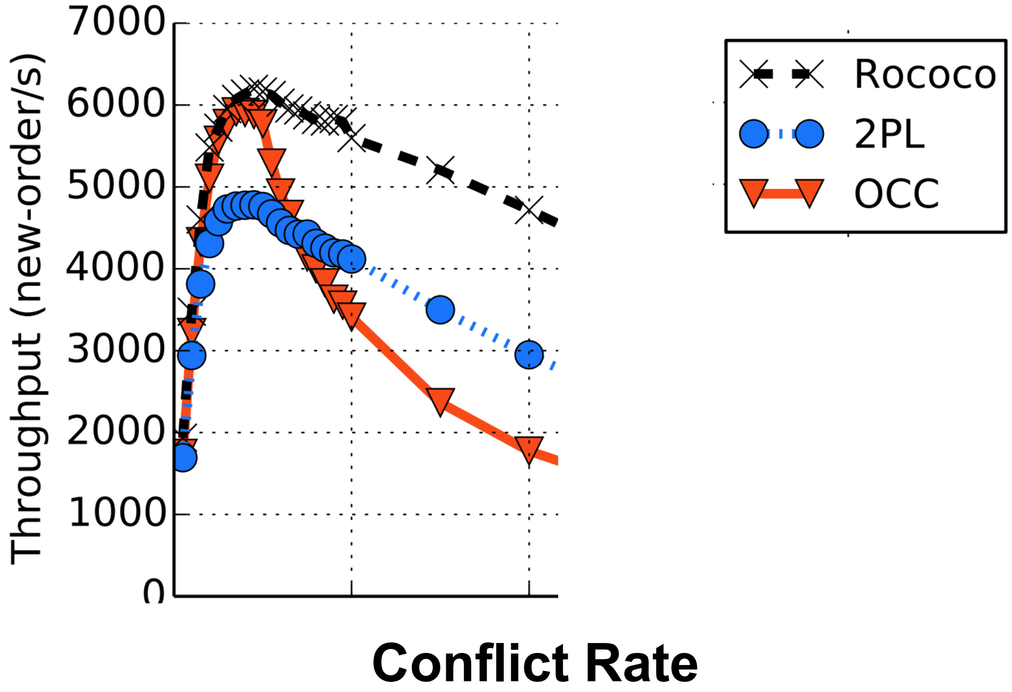
- Acquire locks to **prevent** all possible violations of serializability

- **But leaves a lot of concurrency on the table that is okay and available**

Be optimistic!

- Goal: Low overhead for non-conflicting txns
- Assume success!
  - Process transaction as if it would succeed
  - Check for serializability only at commit time
  - If fails, abort transaction
- Optimistic Concurrency Control (OCC)
  - Higher performance when few conflicts vs. locking
  - Lower performance when many conflicts vs. locking

2PL vs OCC



- From Rococo paper in OSDI 2014. Focus on 2PL vs. OCC.
- Observe OCC better when write rate lower (fewer conflicts), worse than 2PL with write rate higher (more conflicts)

Optimistic Concurrency Control

- Optimistic Execution:
  - Execute reads against shards
  - Buffer writes locally

- Validation and Commit:
  - Validate that data is still the same as previously observed
    - (i.e., reading now would give the same result)
  - Commit the transaction by applying all buffered writes
  - Need this to all happen together, how?

Validation and Commit use 2PC

- Client sends each shard a prepare
  - Prepare includes read values and buffered writes for each shard
  - Each shard acquires shared locks on read locations and exclusive locks on write locks
  - Each shard checks if read values validate
  - Each shard sends vote to client
    - If all locks acquired and reads validate => Vote Yes
    - Otherwise => Vote No

- Client collects all votes, if all yes then commit
  - Client sends commit/abort to all shards
  - If commit: shards apply buffered writes
  - Shards release all locks