GEORGE MASON UNIVERSITY

DBMS

Concurrency Control, Recovery, and Locking

CS 475: Concurrent & Distributed Systems (Fall 2021)

Lecture 14

Yue Cheng

Some material taken/derived from:

- Princeton COS-418 materials created by Michael Freedman and Kyle Jamieson.
- MIT 6.824 by Robert Morris, Frans Kaashoek, and Nickolai Zeldovich.

The transaction

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

All-or-nothing

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

Transaction examples

- Bank account transfer
  - Turing -= $100
  - Lovelace += $100

Banking.

- Maintaining symmetric relationships
  - Lovelace FriendOf Turing
  - Turing FriendOf Lovelace

Social website.

- Order product
  - Charge customer card
  - Decrement stock
  - Ship stock

E_ commerce.

Relationship with replication

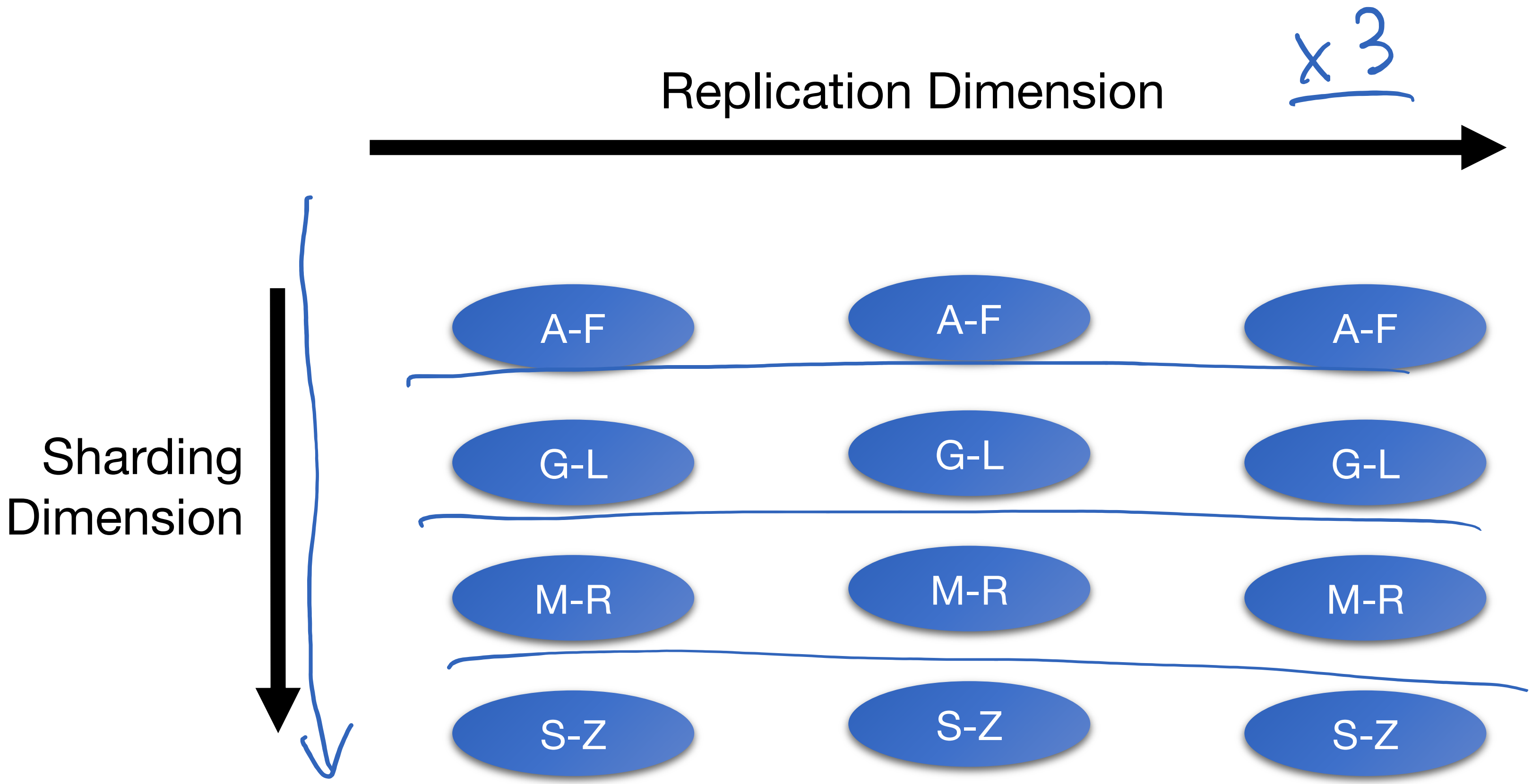
- **Replication** (e.g., Raft) is about doing the **same** thing in multiple places to provide fault tolerance

- **Sharding** is about doing **different** things in multiple places for scalability
  - e.g., using consistent hashing to partition data in distributed storage (Dynamo)

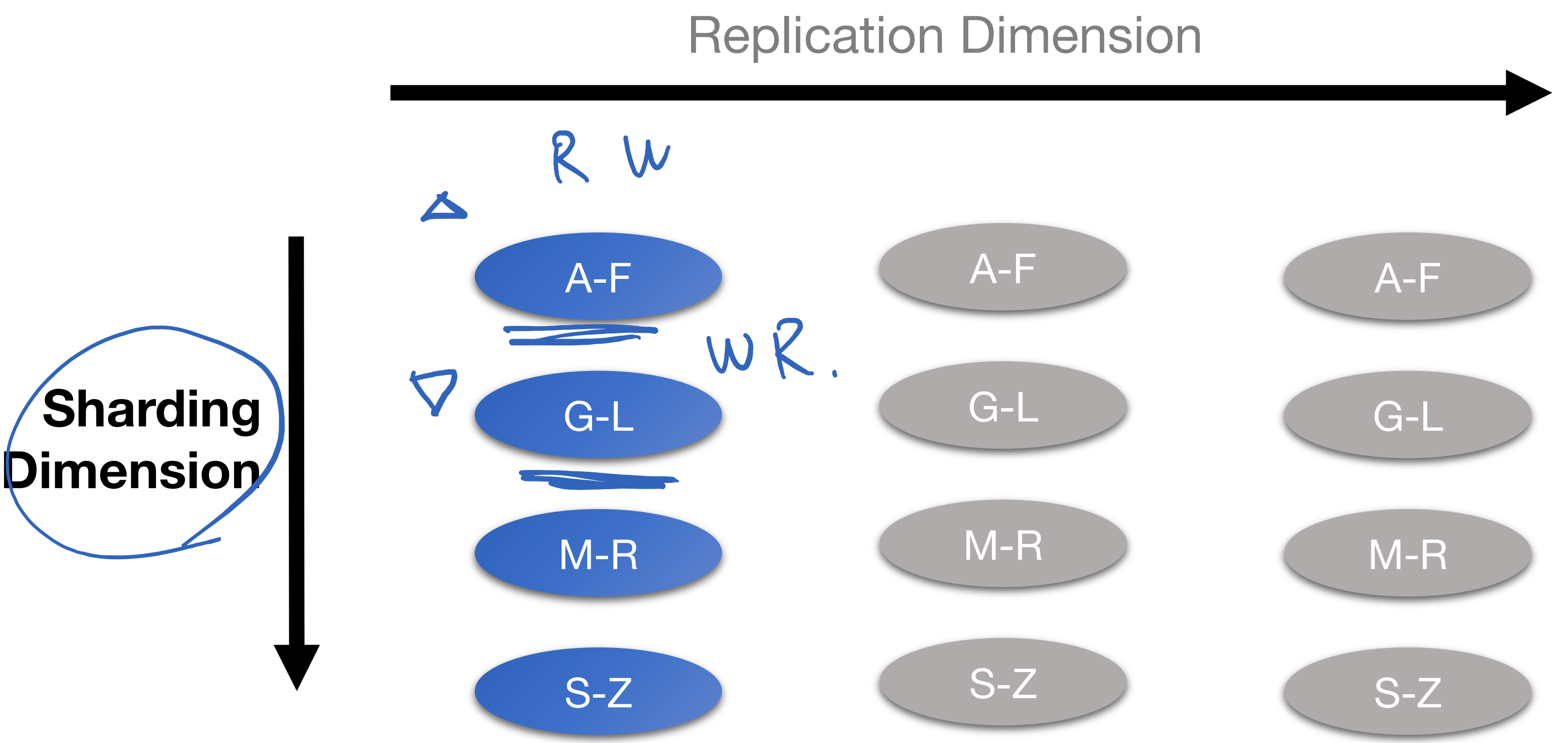
Txn

- **Atomic commit** is about doing **different** things in **different** places together

Relationship with replication



Focus on sharding for today



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

Challenges

1. High transaction **speed requirements**
   - If always `fsync()` to disk for each result on transaction, yields terrible performance

2. **Atomic and durable** writes to disk are difficult
   - In a manner to handle arbitrary crashes
   - Hard disks and solid-state storage use **write buffers** in volatile memory

Outline

Techniques for achieving ACID properties

- Write-ahead logging and checkpointing
- Serializability and two-phase locking

What does the system need to do?

- Transaction's properties: **ACID**
  - Atomicity, Consistency, Isolation, Durability
- **Application logic** checks **consistency** (C)
- This leaves **two main goals** for the **system:**

1. Handle **failures** (**A, D**)
   Availability
   Correctness
2. Handle **concurrency** (**I**)

Goal #1: Concurrency control

Transaction **recovery**

Failure model: crash failures

- Standard "crash failure" model:

- Machines are prone to crashes:
    - Disk contents (*non-volatile storage*) **okay**
    - Memory contents (*volatile storage*) **lost**

- Machines don't misbehave ("Byzantine")

Account transfer transaction

- Transfers $10 from account A to account B

```
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
```

Problem

```
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
```

- Suppose $100 in A, $100 in B

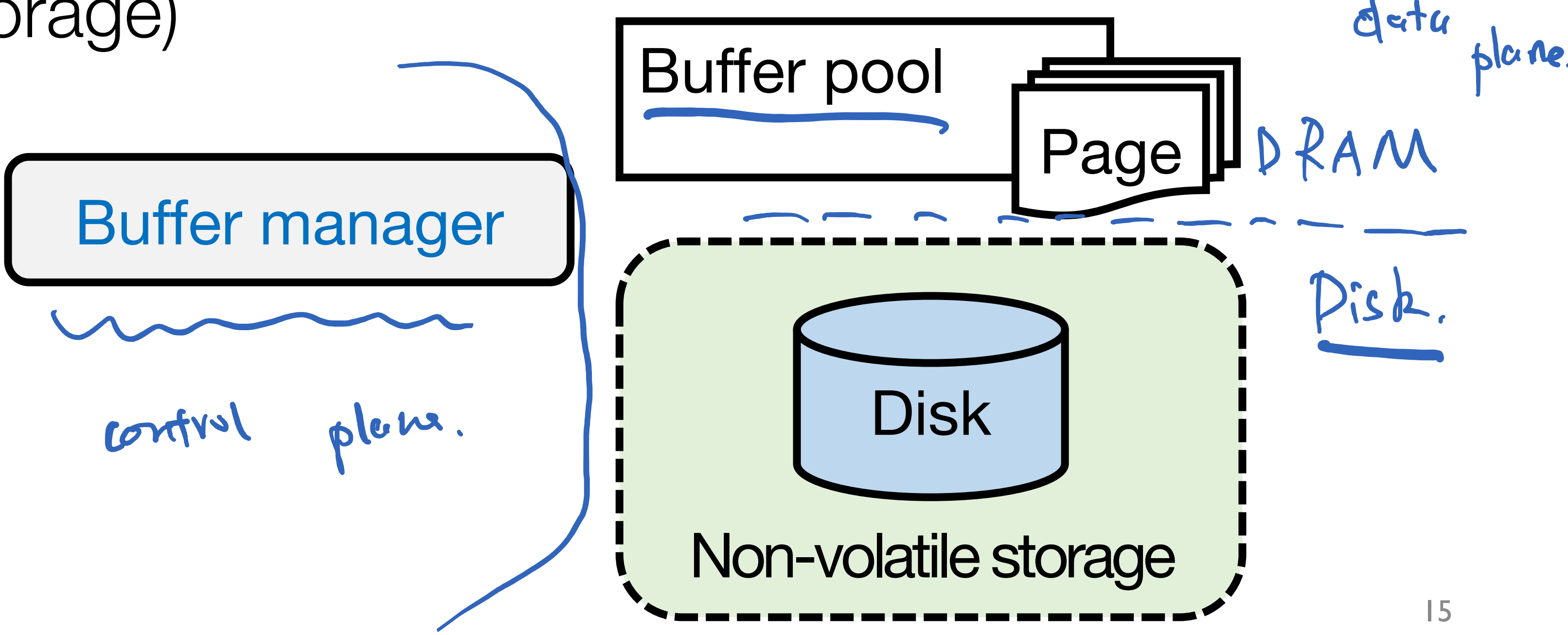
- commit_tx starts the commit protocol:
  - write(A, $90) to disk
  - write(B, $110) to disk

- What happens if system crash **after first write**, but **before second write**?
  - After recovery: Partial writes, money is lost

Lack atomicity in the presence of failures

System architecture

- Smallest unit of storage that can be atomically written to non-volatile storage is called a **page**
- **Buffer manager** moves pages between **buffer pool** (in volatile memory) and disk (in non-volatile storage)



Two design choices

1. **Force** all of a transaction's writes to disk **before** transaction commits?
   - Yes: ***force*** policy
   - No: ***no-force*** policy

2. May **uncommitted** transactions' writes **overwrite** committed values on disk?
   - Yes: ***steal*** policy
   - No: ***no-steal*** policy

Performance implications

1. **Force** all of a transaction's writes to disk **before** transaction commits?
   - Yes: **force** policy

   Then slower disk writes appear on the critical path of a committing transaction

2. May **uncommitted** transactions' writes **overwrite** committed values on disk?
   - No: **no-steal** policy

   Then buffer manager loses write scheduling flexibility

Undo & redo

1. **Force** all a transaction's writes to disk **before** transaction commits?
   - Choose **no: no-force** policy
     - ☞ **Need support for** *redo:* complete a committed transaction's writes on disk

2. May **uncommitted** transactions' writes **overwrite** committed values on disk?
   - Choose **yes: steal** policy
     - ☞ **Need support for** *undo:* removing the effects of an uncommitted transaction on disk

How to implement undo & redo?

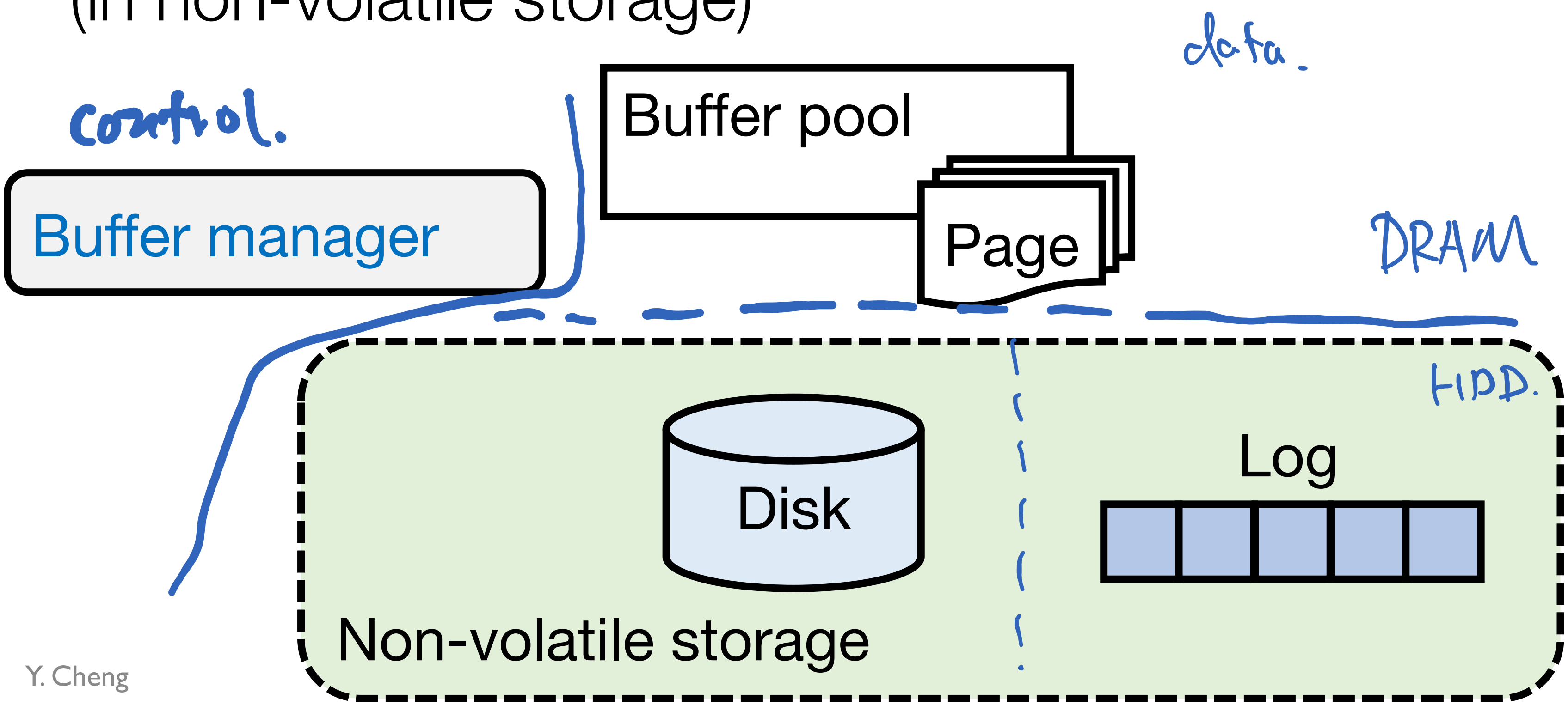
- **Log:** A sequential file that stores information about transactions and system state
  - Resides in separate, non-volatile storage

- One entry in the log for each update, commit, abort operation: called a log record

- Log record contains:
  - Monotonic-increasing log sequence number (LSN)
  - **Old value** (before image) of the item for **undo** steal
  - **New value** (after image) of the item for **redo** no-force.

System architecture

- **Buffer pool** (volatile memory) and disk (non-volatile)
- The **log** resides on a **separate** partition or disk (in non-volatile storage)



Write-ahead logging (WAL)

- Ensures atomicity in the event of system crashes under **no-force/steal** buffer management

1. Force all log records pertaining to an updated page into the (non-volatile) log before any (over)-writes **to page itself** → Ensure Undo info → (Steal).
2. A transaction is not considered committed until **all its log records** (including commit record) are forced into the log → Ensure Redo info → (No-force).

WAL example

Undo. Redo.

- force_log_entry(A, old=$100, new=$90)
- force_log_entry(B, old=$100, new=$110)
- write(A, $90)
- write(B, $110)
- force_log_entry(commit)

Does **not** have to flush to disk

- What if the commit log record size > the page size?
- How to ensure **each log record** is written atomically?
  - Write a checksum of entire log entry

Goal #2: Concurrency control

Transaction **isolation**

Two concurrent transactions

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

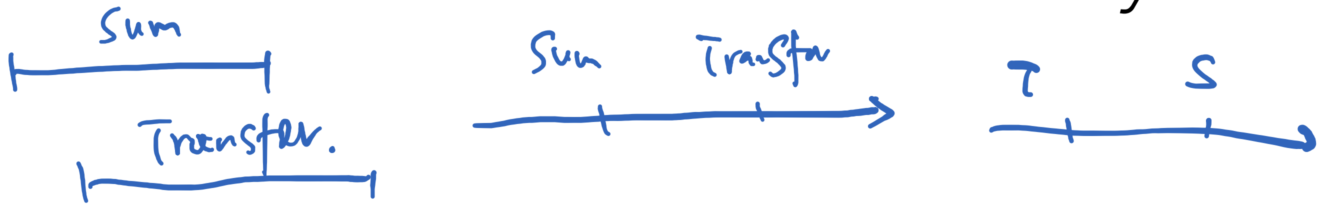
Read Only.

```
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
```

R/W.

Isolation between transactions

- **Isolation:** sum appears to happen either completely before or completely after **transfer**
  - 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:

debit | credit

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:

debit | credit

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**
  - 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

* from two txns.
* same data.
* at least one op → W

- 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**

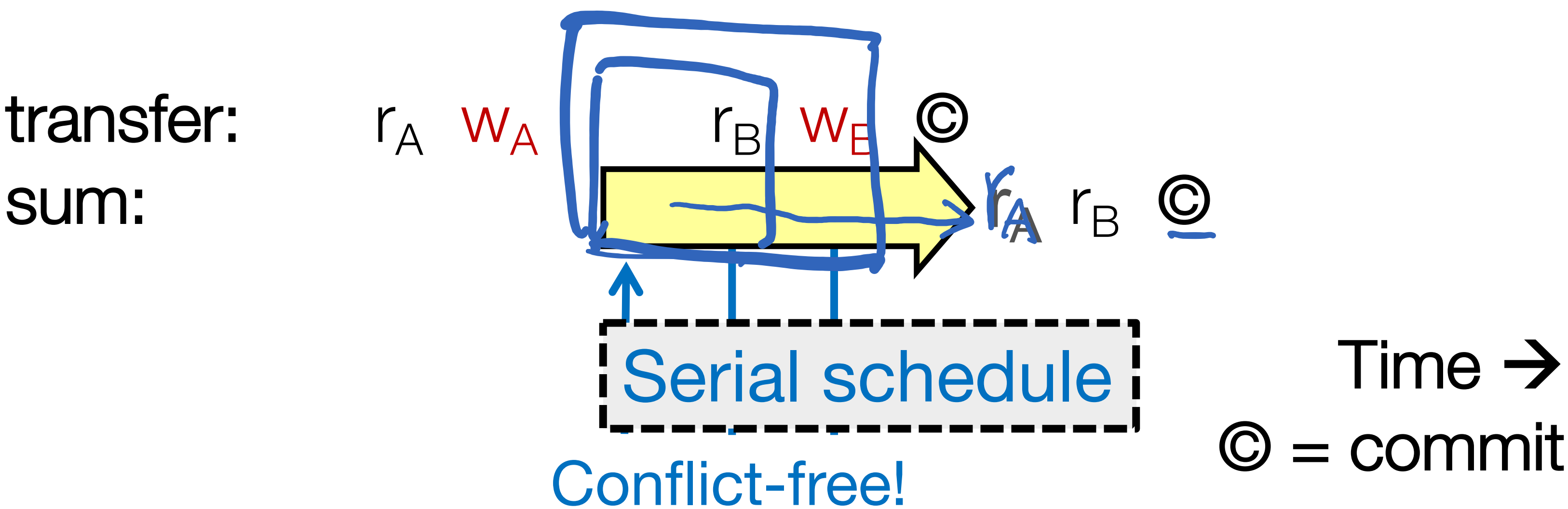
Conflict serializability

- Ideal isolation semantics: *conflict serializability*

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



A non-serializable schedule

- Ideal isolation semantics: *conflict serializability*
- A schedule is ***conflict 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 ©

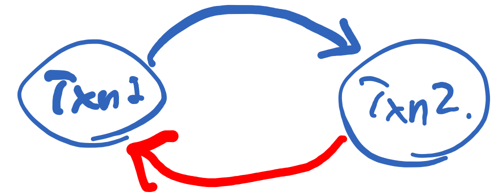
But in a serial schedule, sum's reads either both before w_A or both after w_B

Conflicting ops

Time →

© = commit

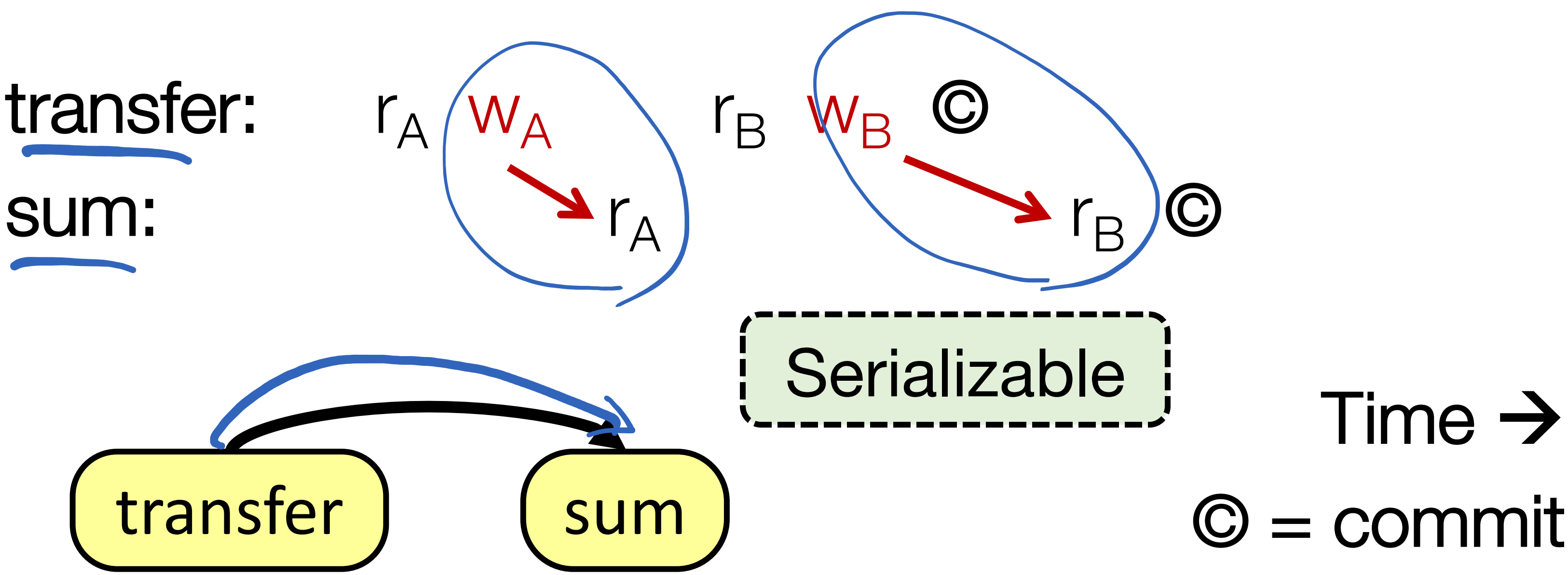
Testing for serializability



- 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*

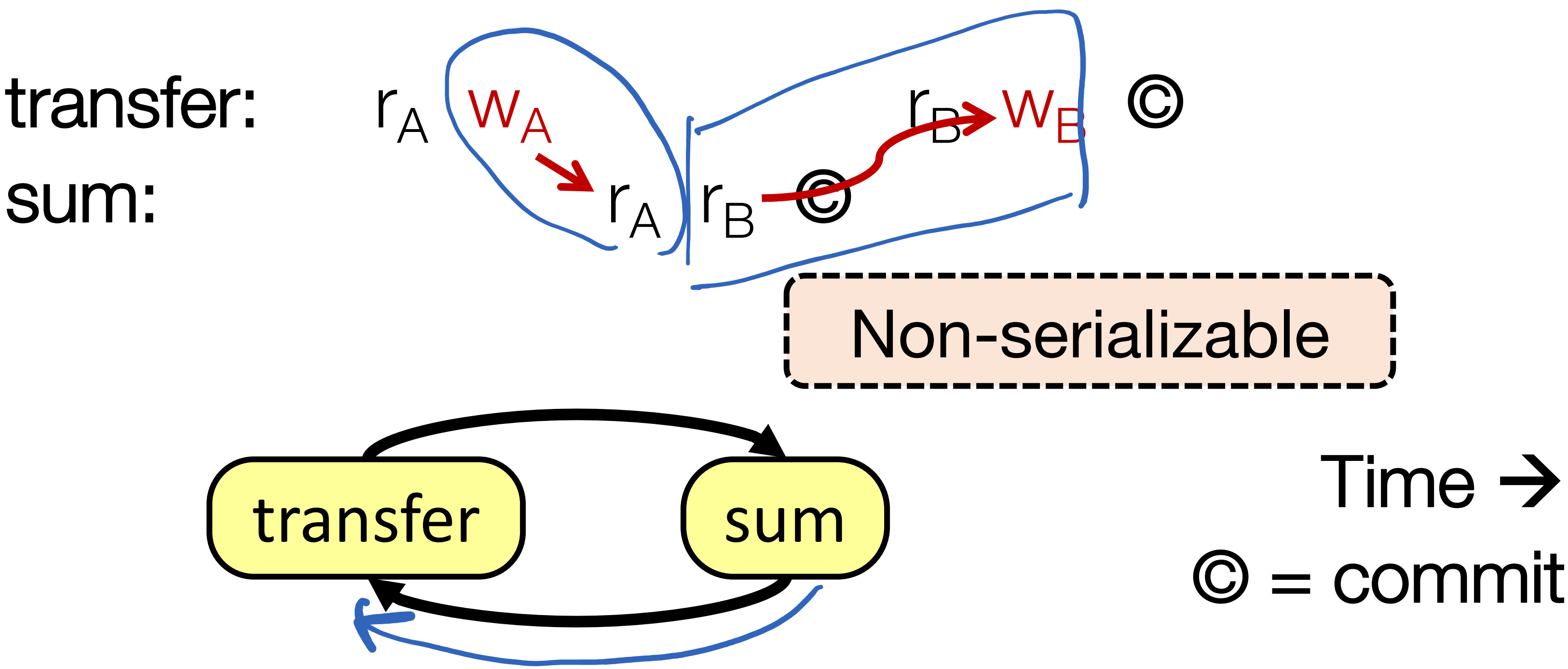
Serializable schedule, acyclic graph

- 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*



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*



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 conflict-serializable if and only if its precedence graph is acyclic

How to ensure a serializable schedule?

- Locking-based approaches

- Strawman 1: Big global lock
  - Acquire the lock when transaction starts
  - Release the lock when transaction ends

Results in a *serial* transaction schedule at the cost of performance

Locking

- Locks maintained by transaction manager
  - Transaction requests lock for a data item
  - Transaction manager **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

How to ensure a serializable schedule?

- Strawman 2: Grab (fine-grained) locks **independently**, for each data item (*e.g.*, bank accounts A and B)

XL(A) XU(A) SL(B) XL(B) XU(B)

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

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

SL(A) SU(A) SU(B)

Permits this non-serializable interleaving

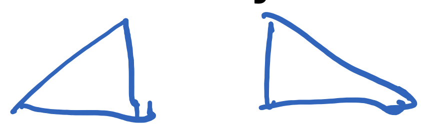
Time →

© = 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



- A **growing phase** when txn acquires locks
- A **shrinking phase** when txn releases locks

- In practice: Strict 2PL
  - Growing phase is the entire transaction
  - Shrinking phase is during commit

2PL allows only serializable schedules

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

Ux(A).

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

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

Us(A).

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

UX(A)

transfer: r_A w_A r_B w_B ©

sum: r_A r_B ©

LS(A).

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

Time →

© = commit

(locking not shown)

Issues with 2PL

waits-for graph

- What if a lock is unavailable? Is deadlock possible?
  - Yes; but a central controller can detect deadlock cycles and abort involved transactions

- The phantom problem row.
  - Database has fancier ops than key-value store
  - T1: begin_tx; update employee (set salary = 1.1×salary) where dept = “CS”; commit_tx
  - T2: insert into employee (“carol”, “CS”)
    - Even if they lock individual data items, could result in non-serializable execution

Linearizability vs. Serializability

- **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

- Linearizability + serializability = **strict serializability**
  – Transaction behavior equivalent to some serial execution
    - And that serial execution agrees with real-time

Summary

Techniques for achieving ACID properties

- Write-ahead logging and check-pointing → A, D
- Serializability and two-phase locking → I