

Concurrency Control, Locking, and Recovery



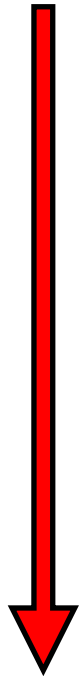
جامعة الملك عبد الله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: Computing Systems and Concurrency Lecture 17

Marco Canini

Credits: Michael Freedman and Kyle Jamieson developed much of the original material.
Selected content adapted from A. LaPaugh, J. Li.

Failures in complex systems propagate



- Say **one bit** in a DRAM fails:
- ...flips a bit in a kernel memory write
- ...causes a **kernel panic,**
- ...program is running an NFS server,
- ...a client **can't read from FS,** so hangs

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 database are made permanent, visible to other transactions
 - When **abort**, database restored to a state such that the aborting transaction never executed

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

Today

- 1. Techniques for achieving ACID properties**
 - Write-ahead logging and checkpointing
 - Serializability and two-phase locking
- 2. Algorithms for Recovery and Isolation Exploiting Semantics (ARIES)**

What does the system need to do?

- Transactions properties: **ACID**
 - Atomicity, Consistency, Isolation, Durability
- **Application logic** checks **consistency (C)**
- This leaves **two main goals** for the **system**:
 1. Handle **failures (A, D)**
 2. Handle **concurrency (I)**

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

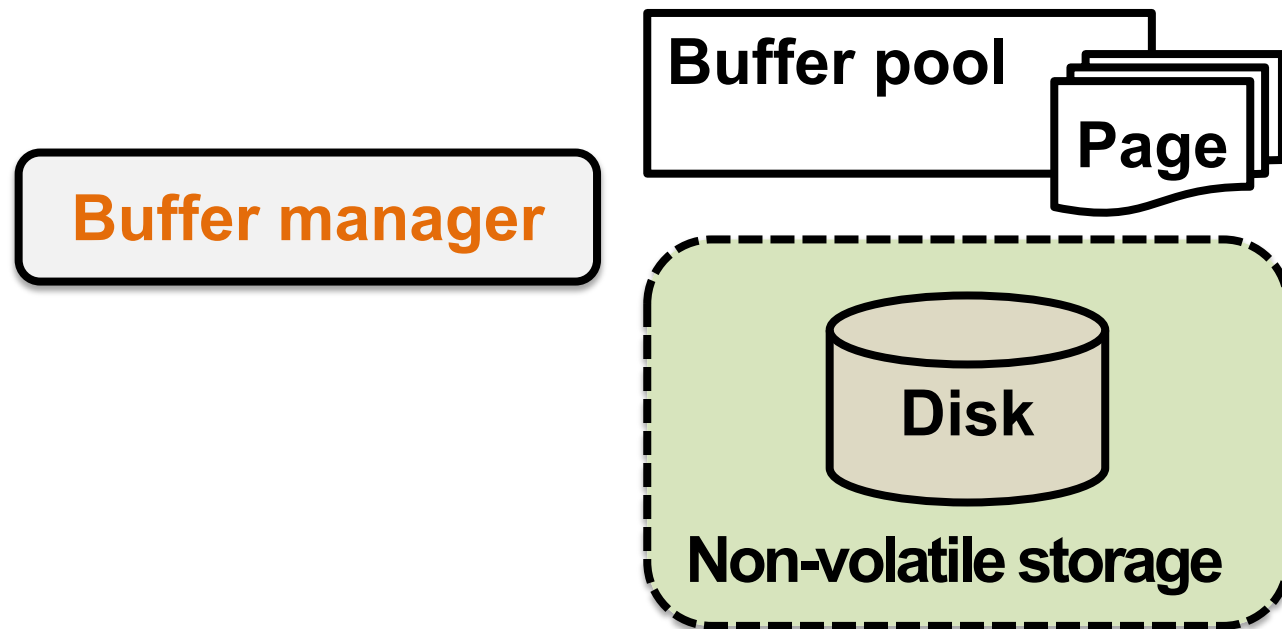
- Suppose \$100 in A, \$100 in B
- `commit_tx` starts the commit protocol:
 - `write(A, $90)` to disk
 - `write(B, $110)` to disk
- What happens if **system crash** after first write, but **before second write**?
 - After recovery: Partial writes, **money is lost**

```
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
```

Lack atomicity in the presence of failures

System structure

- 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 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 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

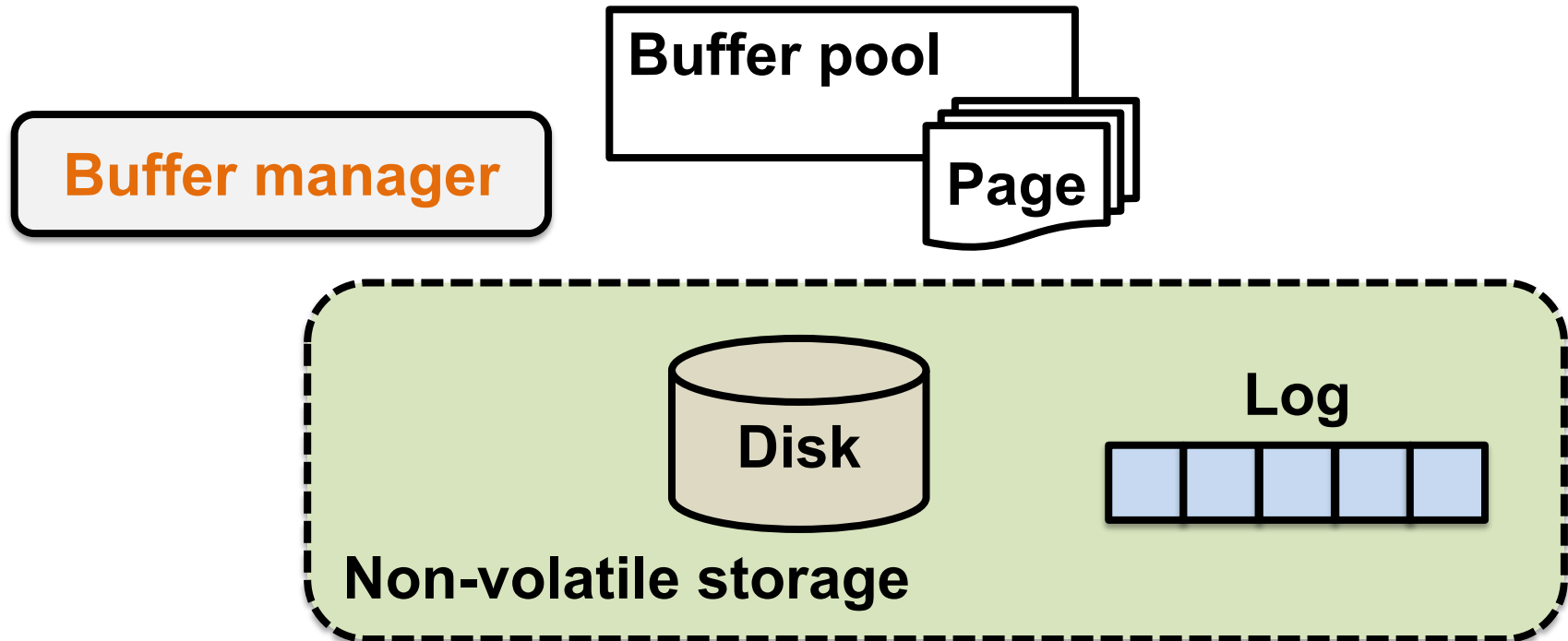
1. **Force** all a transaction's writes to disk **before** transaction commits?
 - Choose **no: no-force** policy
 - 👉 **Need support for *redo***: complete a committed transaction's writes on disk
2. May **uncommitted** transactions' writes **overwrite** committed values on disk?
 - Choose **yes: steal** 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**
 - **New value** (**after image**) of the item for **redo**

System structure

- **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 writes to page itself**
 2. A transaction is not considered committed until **all its log records** (including commit record) are **forced into the log**

WAL example

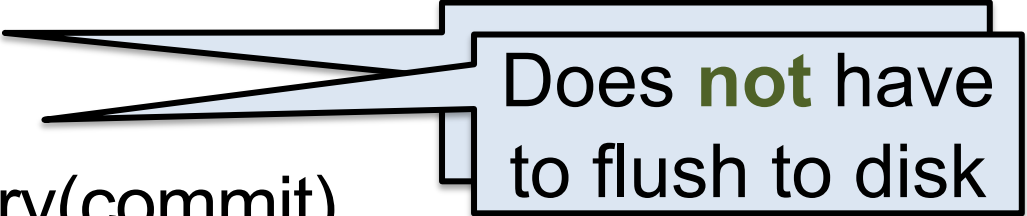
force_log_entry(A, old=\$100, new=\$90)

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
```

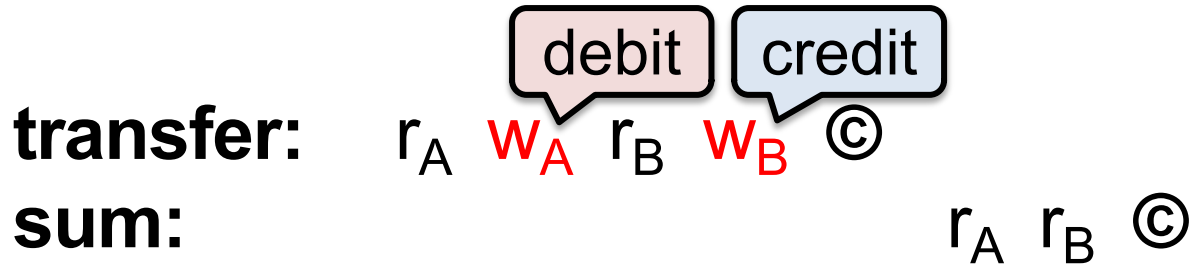
```
transaction transfer(A, B):  
begin_tx  
a ← read(A)  
if a < 10 then abort_tx  
else write(A, a-10)  
b ← read(B)  
write(B, b+10)  
commit_tx
```

Isolation between transactions

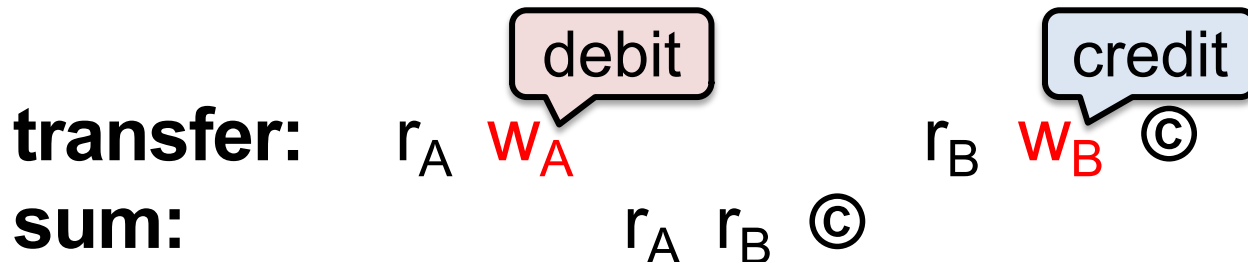
- **Isolation:** **sum** appears to happen either completely before or completely after **transfer**
 - Sometimes called *before-after atomicity*
- *Schedule* for transactions is an ordering of the operations performed by those transactions

Problem for concurrent execution: Inconsistent retrieval

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



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



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

- 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**

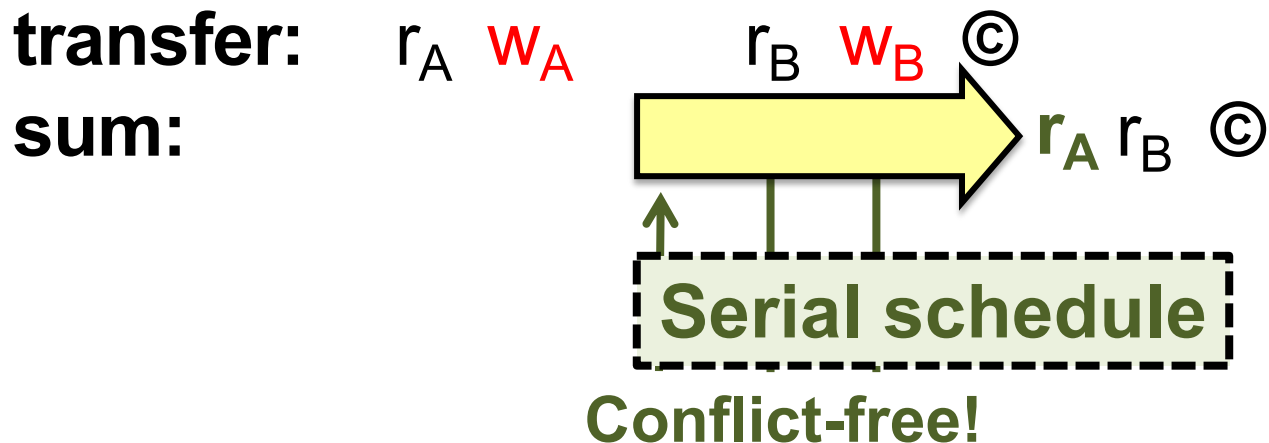
- Two **schedules** are ***equivalent*** if:
 1. They contain the same transactions and operations
 2. They **order** all **conflicting** operations of non-aborting transactions in the **same way**

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



Time →
© = commit

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

Time →
© = **commit**

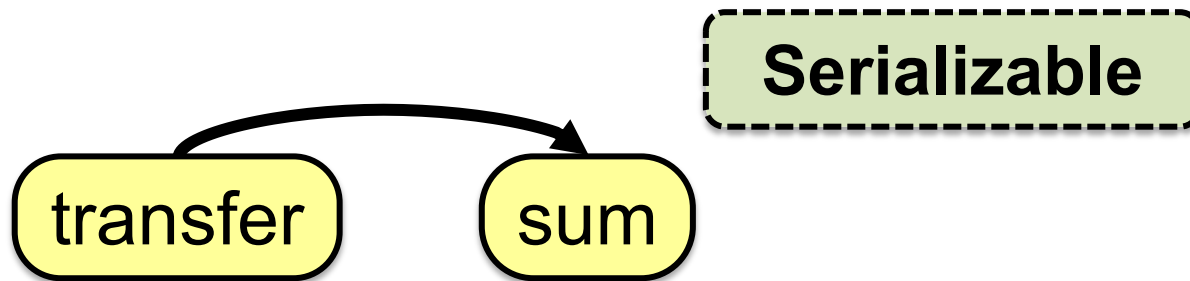
Testing for serializability

- 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

transfer: r_A W_A r_B W_B ©
sum: r_A r_B ©

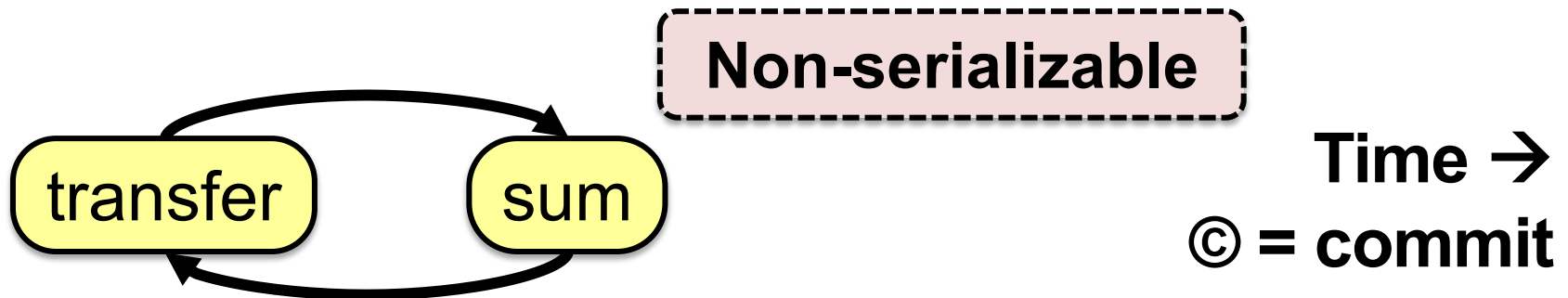


© = commit
Time →

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 ©



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



Permits this **non-serializable** interleaving

Time \rightarrow

\textcircled{C} = commit

$\blacktriangleleft / \triangleleft$ = eXclusive- / Shared-lock; $\blacktriangleright / \triangleright$ = 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 transaction acquires locks
- A **shrinking phase** when transaction releases locks
- In practice:
 - 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



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

Time →

© = commit

$\blacktriangleleft / \triangleleft = X- / S\text{-lock}; \blacktriangleright / \triangleleft = X- / S\text{-unlock}$

2PL and transaction concurrency

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

transfer: $\triangleleft_A r_A$ $\triangleleft_A W_A$ $\triangleleft_B r_B$ $\triangleleft_B W_B * \textcircled{C}$
sum: $\triangleleft_A r_A$ $\triangleleft_B r_B * \textcircled{C}$

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

Time \rightarrow

\textcircled{C} = commit

$\triangleleft / \triangle = X- / S\text{-lock}$; $\blacktriangleright / \blacktriangleright = X- / S\text{-unlock}$; $*$ = 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 if a lock is unavailable? Is **deadlock** possible?
 - Yes; but a central controller can detect deadlock cycles and **abort involved transactions**
- The **phantom problem**
 - Database has fancier ops than key-value store
 - T1: begin_tx; update employee (set salary = $1.1 \times$ 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**

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

Today

1. Techniques for achieving ACID properties
 - Write-ahead logging and check-pointing → **A,D**
 - Serializability and two-phase locking → **I**
2. **Algorithms for Recovery and Isolation Exploiting Semantics (ARIES)**

ARIES (Mohan, 1992)

- In IBM DB2 & MSFT SQL Server, gold standard
- Key ideas:
 1. Refinement of WAL (steal/no-force buffer management policy)
 2. Repeating history after restart due to a crash (*redo*)
 3. Log every change, *even undo operations during crash recovery*
 - Helps for repeated crash/restarts

ARIES' stable storage data structures

- Log, composed of log records, each containing:
 - **LSN**: Log sequence number (monotonic)
 - **prevLSN**: Pointer to the previous log record for the same transaction
 - A linked list for each transaction, “threaded” through the log
- Pages
 - **pageLSN**: Uniquely identifies the log record for the latest update applied to this page

ARIES' in-memory data structures

- **Transaction table** (*T-table*): one entry per transaction
 - Transaction identifier
 - Transaction status (running, committed, aborted)
 - **lastLSN**: LSN of the most recent **log record** written by the transaction
- **Dirty page table**: one entry per page
 - Page identifier
 - **recoveryLSN**: LSN of log record for **earliest** change to that page **not on disk**

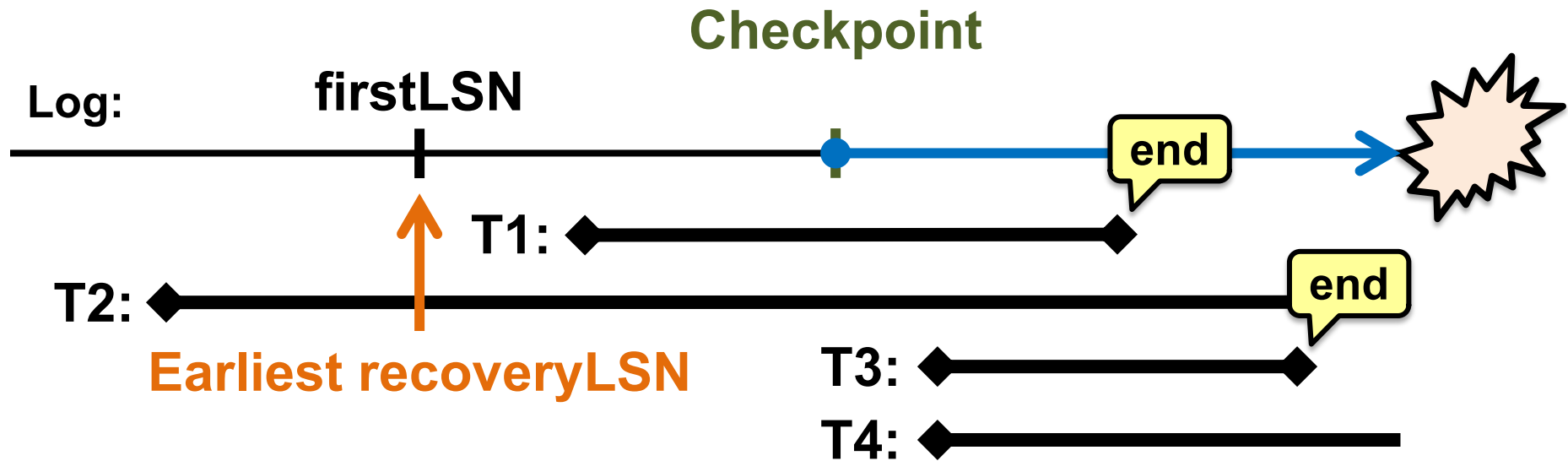
Transaction commit

1. Write **commit** log record to the (non-volatile) log
 - Signifies that the commit is **beginning** (it's not the actual commit point)
2. Write all log records associated with this transaction to the log
3. Write **end** log record to the log
 - This is the **actual “commit point”**

Checkpoint

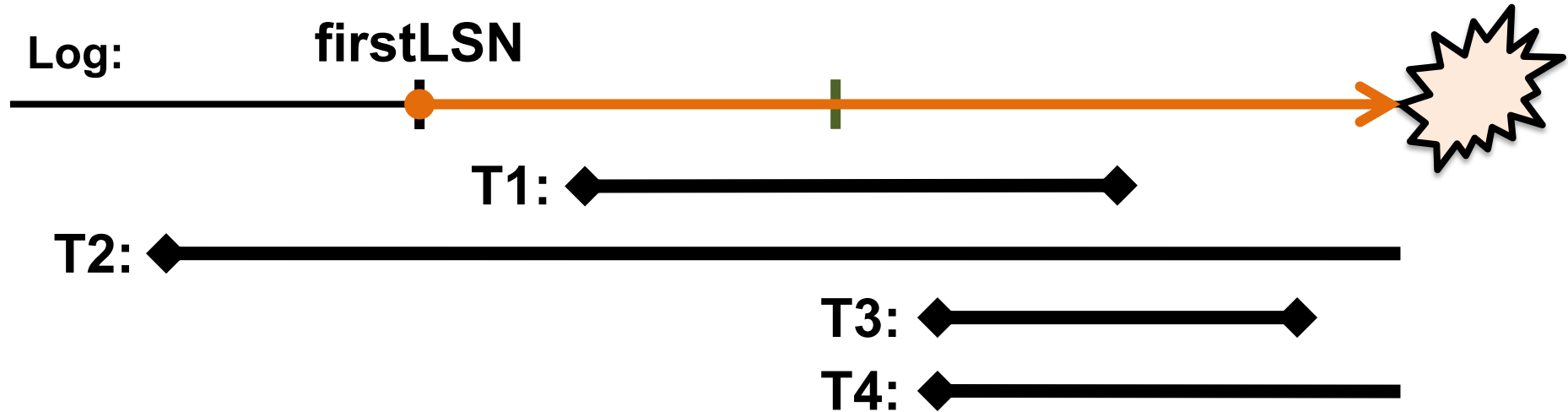
- Happens while other transactions are running, as a separate transaction
 - **Does not flush dirty pages** to disk
 - **Does** tell us **how much to fix** on crash
1. Write **“begin checkpoint”** to log
 2. Write current **transaction table, dirty page table,** and **“end checkpoint”** to log
 3. Force log to non-volatile storage
 4. Store “begin checkpoint” LSN → **master record**

Crash recovery: Phase 1 (Analysis)



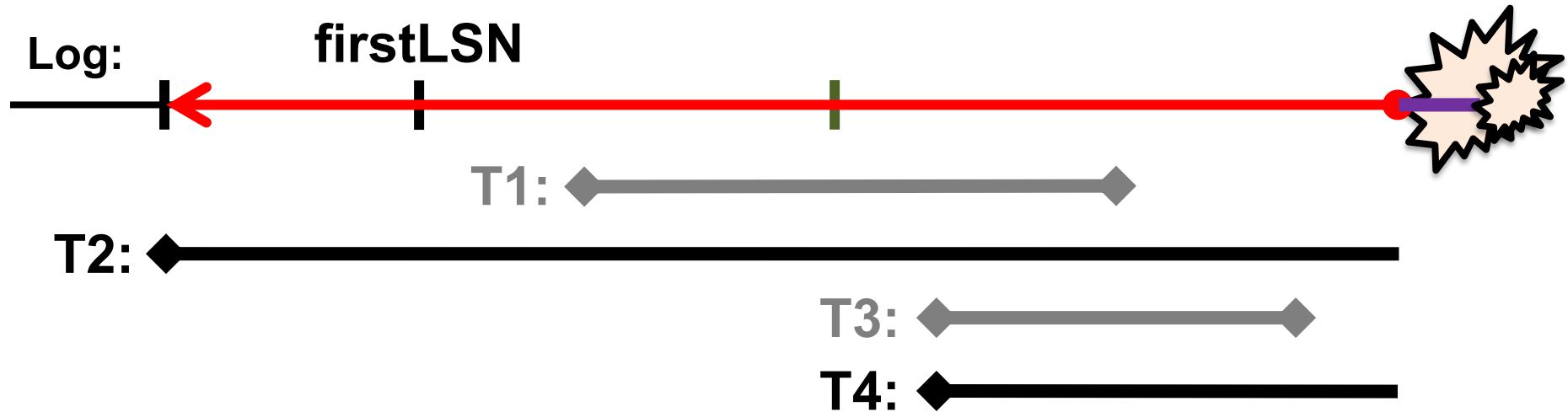
1. Start with **checkpointed** T- & dirty page-tables
2. Read log **forward from checkpoint**, updating tables
 - For **end** entries, remove T from T-table (T1, T3)
 - For other log entries, add (T2, T4) or update T-table
 - Add LSN to dirty page table's **recoveryLSN**

Crash recovery: Phase 2 (REDO)



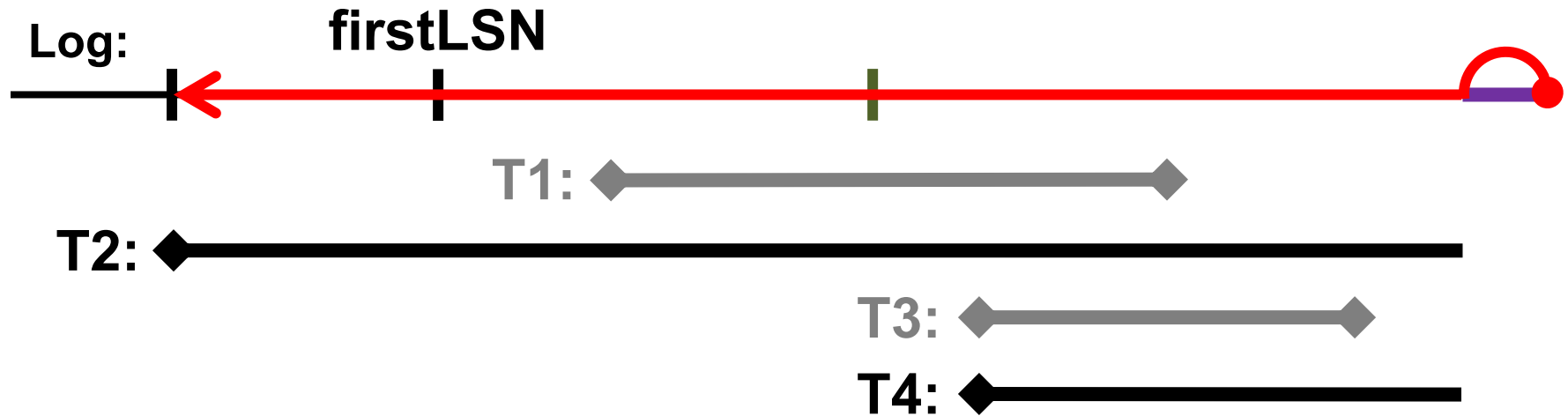
- Start at **firstLSN**, scan log entries forward in time
 - Reapply action, update pageLSN
- Database state now matches state as recorded by log at the time of crash

Crash recovery: Phase 3 (UNDO)



- Scan log entries backwards from the end. For updates:
 - Write *compensation log record* (**CLR**) to log
 - Contains prevLSN for update: **UndoNextLSN**
 - Undo the update's operation

Crash recovery: Phase 3 (UNDO)



- Scan log entries backwards from the end. For **CLRs**:
 - If UndoNextLSN = null, write **end record**
 - Undo for that transaction is done
 - Else, **skip to UndoNextLSN** for processing
 - Turned the undo into a redo, done in Phase 2

ARIES: Concluding thoughts

- Brings together all the concepts we've discussed for ACID, concurrent transactions
- Introduced redo for “repeating history,” novel undo logging for repeated crashes
- For the interested: Compare with ***System R*** (not discussed in this class)

Next topic:

More CC and Distributed Transactions