Transaction Management & Concurrency Control

CS 377: Database Systems

Example: Need for Control

- ATM where a customer has some amount of money in his checking account and wants to withdraw $25

  ```
  READ(A);
  CHECK(A > 25);
  PAY(25);
  A = A - 25;
  WRITE(A);
  ```

- What happens if DBMS crashes right after paying?

- What if his wife also withdraws money at the same time?

Transaction Management

- Inconsistencies can occur when:
  - System crashes, user aborts, …
  - Interleaving actions of different user programs
- Want to provide the users an illusion of a single-user system
  - Why not just allow one user at a time?

Transaction

- A collection of operations that form a single atomic logical unit of execution
  BEGIN TRANSACTION
    <SQL COMMAND>
  END TRANSACTION

- Operations: READ(X) - retrieval, WRITE(X) - insert, delete, update

- Transactions must leave the database in a consistent state

ACID: Transaction Properties

- Atomicity: a transaction is an atomic unit of data processing
  - All actions in transaction happen or none happen
- Consistency: a database in a consistent state will remain in a consistent state after the transaction
  - Any data written to the database must be valid according to constraints, cascades, triggers, etc.

ACID: Transaction Properties (2)

- Isolation: the execution of one transaction is isolated from other transactions
  - Execution of a transaction should not be interfered with by other transactions executing at same time
- Durability: if a transaction commits, its effects must persist
  - Changes should not be lost because of possible failure occurring immediately after transaction

Transaction Management Overview

- Recovery (Atomicity & Durability)
    - Ensures database is fault tolerant, and not corrupted by software, system or media
    - 24x7 access to critical data
- Concurrency control (Isolation)
    - Provide correct and highly available data access in the presence of access by many users
- Application program for consistency

Transaction Terminology

- Commit: successful completion of a transaction — operations of transaction are guaranteed to be performed on the data in the database
- Abort: unsuccessful termination of a transaction —operations of transaction are guaranteed to not be performed on the data in the database
- Rollback: process of undoing updates made by operations of a transaction
- Redo: process of performing the updates made by the operations of a transaction again

SQL Transactions

- A new transaction starts with the BEGIN command (or begins implicitly when a statement is executed)
- Transaction stops with either COMMIT, ABORT, ROLLBACK
  - COMMIT means all changes are saved
  - ABORT means all changes are undone
  - ROLLBACK undoes transactions not already saved

Recovery via System Logs

Idea: Keep a system log and perform recovering when necessary

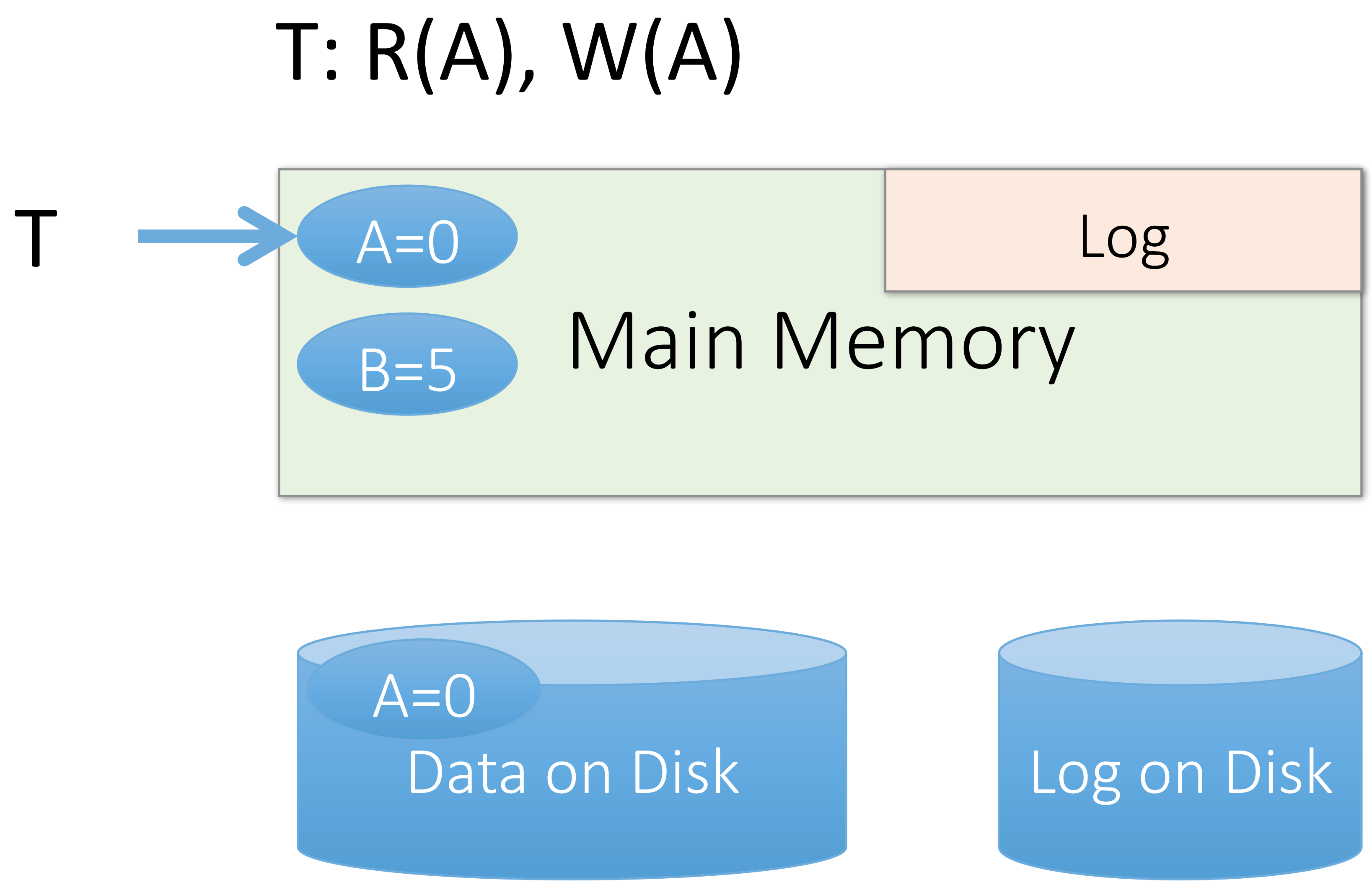
- Separate and non-volatile (stable) storage that is periodically backed up
- Contains log records that contains information about an operation performed by transaction
- Each transaction is assigned a unique transaction ID to different themselves

Logging: Basic Idea

- Record information for every update
  - Sequential writes to log
  - Minimal information written to log
- Used by all modern systems
  - Audit trail & efficiency reasons
- Alternative to logging is shadow paging: make copies of pages and make changes to these copies — only on commit are they made visible to others

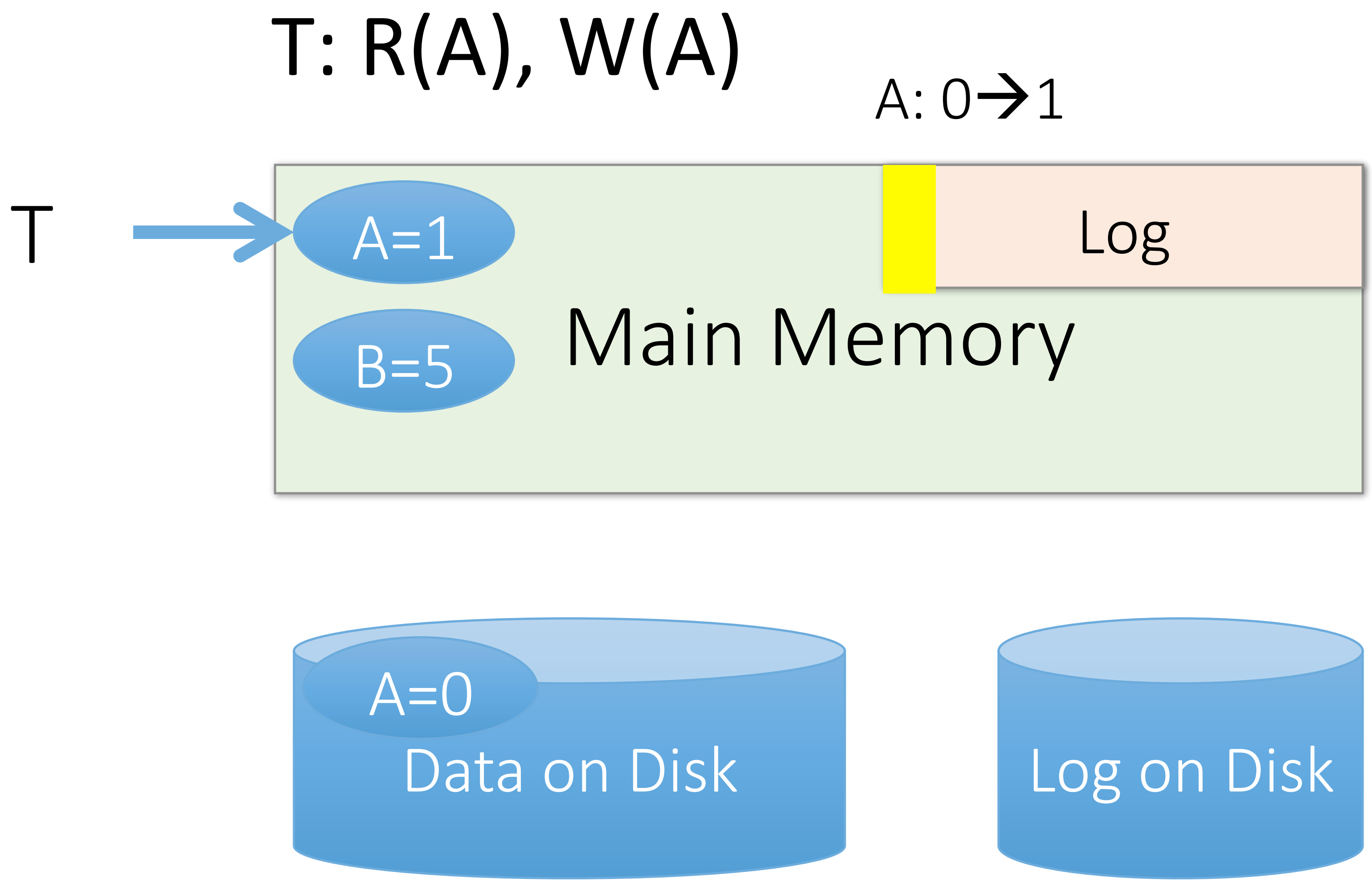
Logging: WAL Picture

Write ahead logging (WAL): all modifications are written to a log before they are applied to database



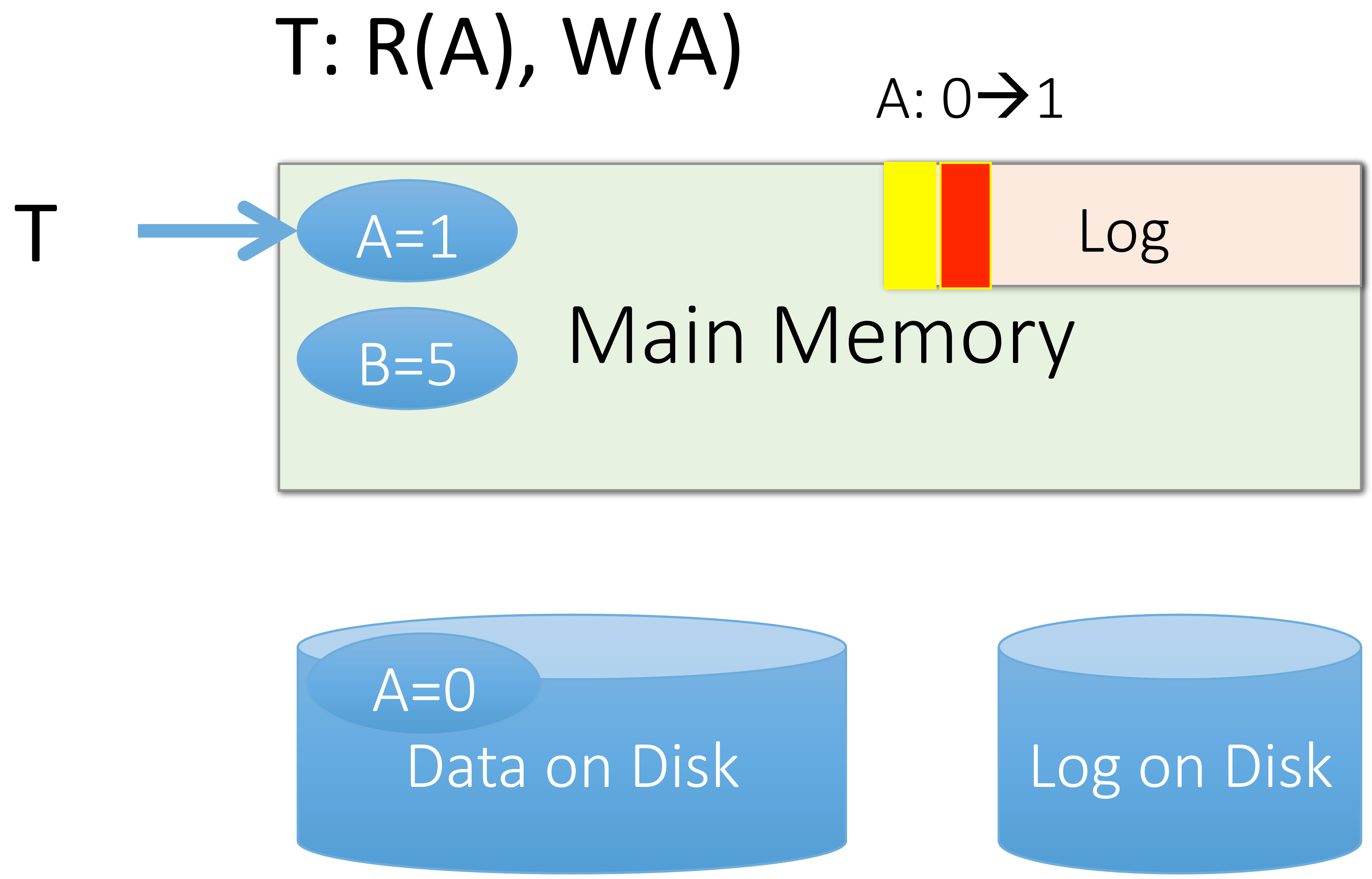
Logging: WAL Picture (2)

Write ahead logging (WAL): all modifications are written to a log before they are applied to database



Logging: WAL Picture (3)

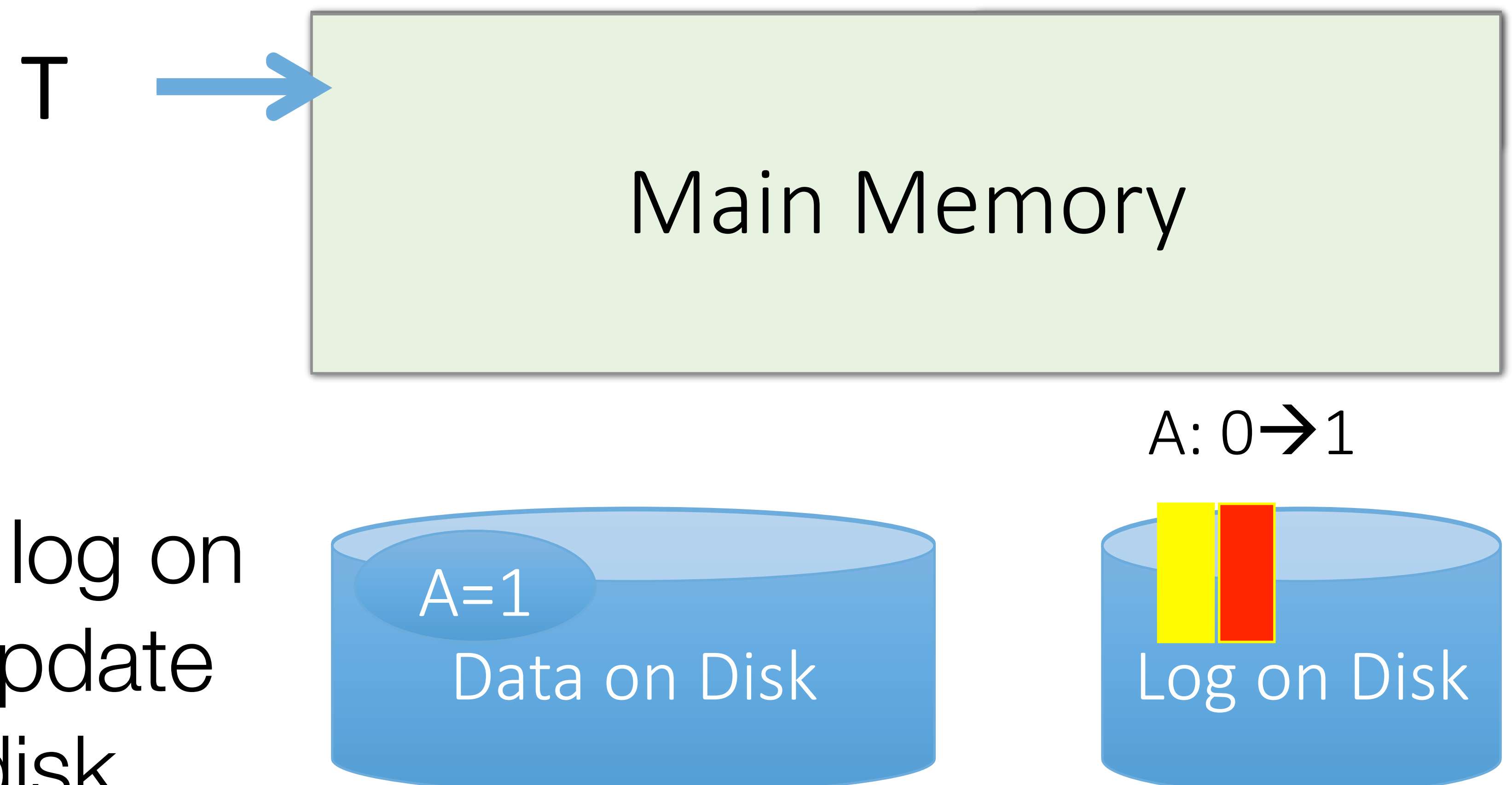
Write ahead logging (WAL): all modifications are written to a log before they are applied to database



Logging: WAL Picture (4)

Write ahead logging (WAL): all modifications are written to a log before they are applied to database

T: R(A), W(A)



First write to log on disk, then update data on disk

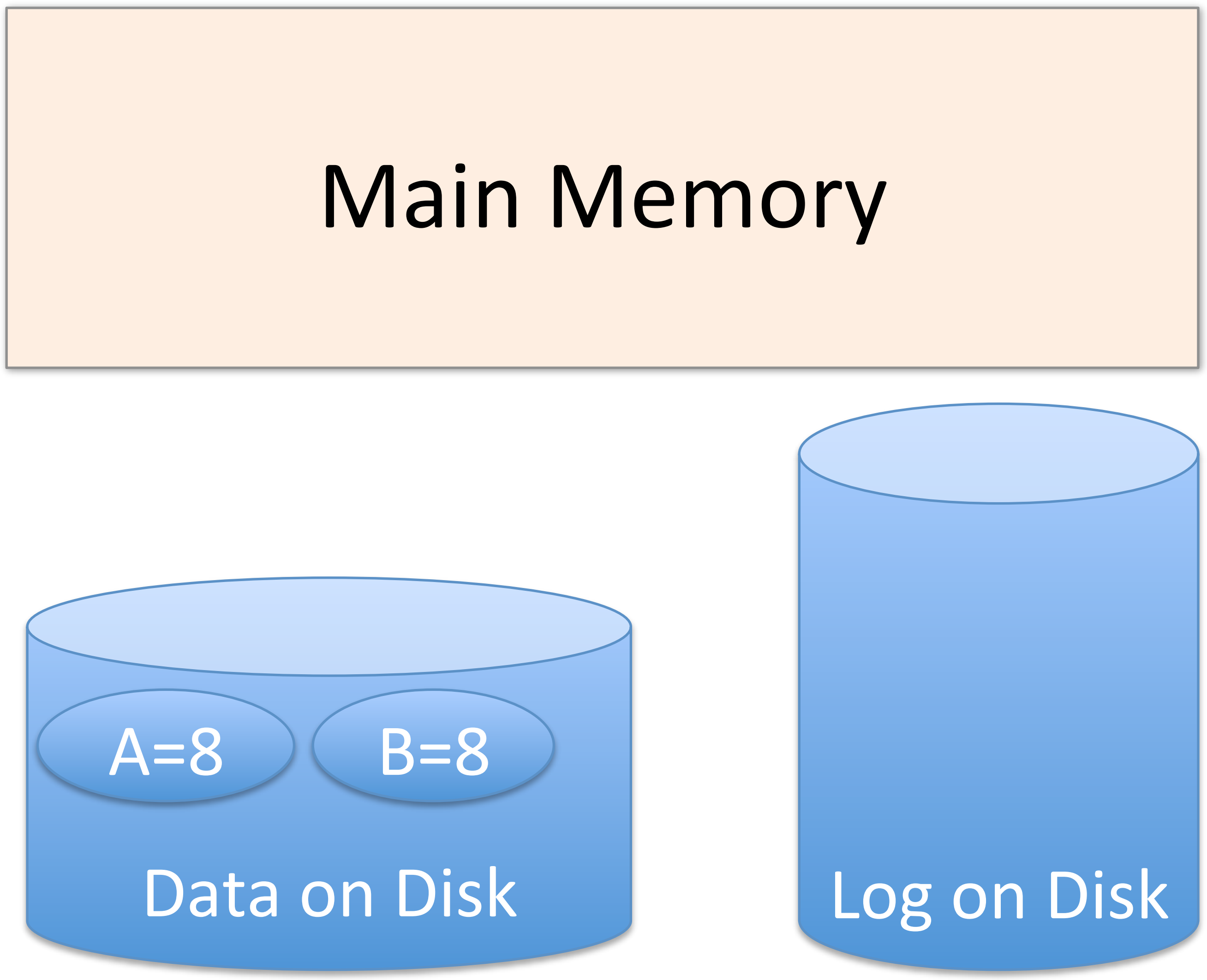
Undo Logging

Idea: undo operations for uncommitted transactions to go back to original state of database

- New transaction begins — add [start, T] to the log
- Read data — do nothing
- Write data — add [write, T, X, old_value], after successful write to log, update X with new value
- Complete transaction — add [commit, T] to log
- Abort transaction — add [abort, T] to log

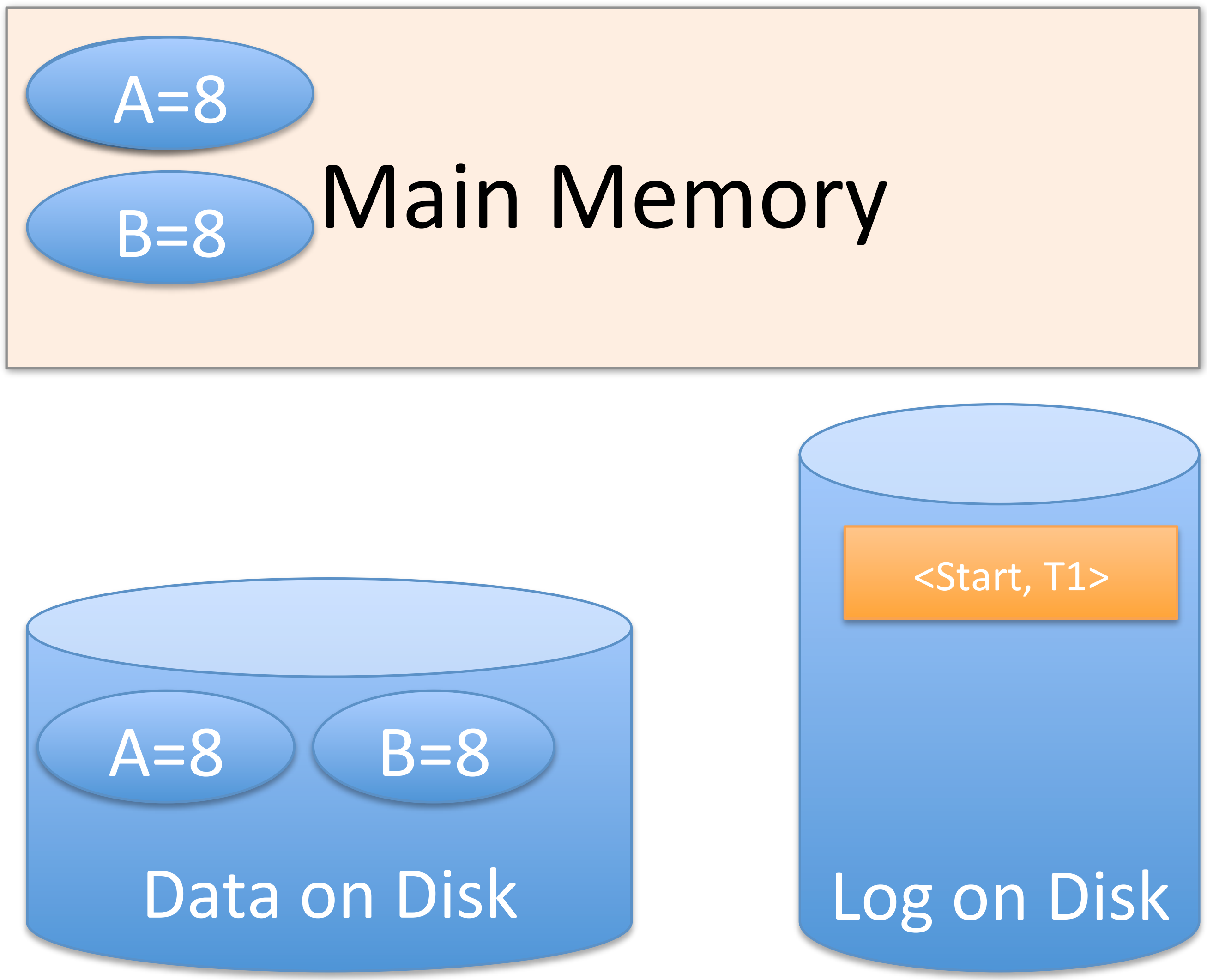
Example: Undo Logging

T1: Read (A, t);
t <— t x 2;
Write(A, t);
Read (B, t);
t <— t x 2;
Write(B, t);



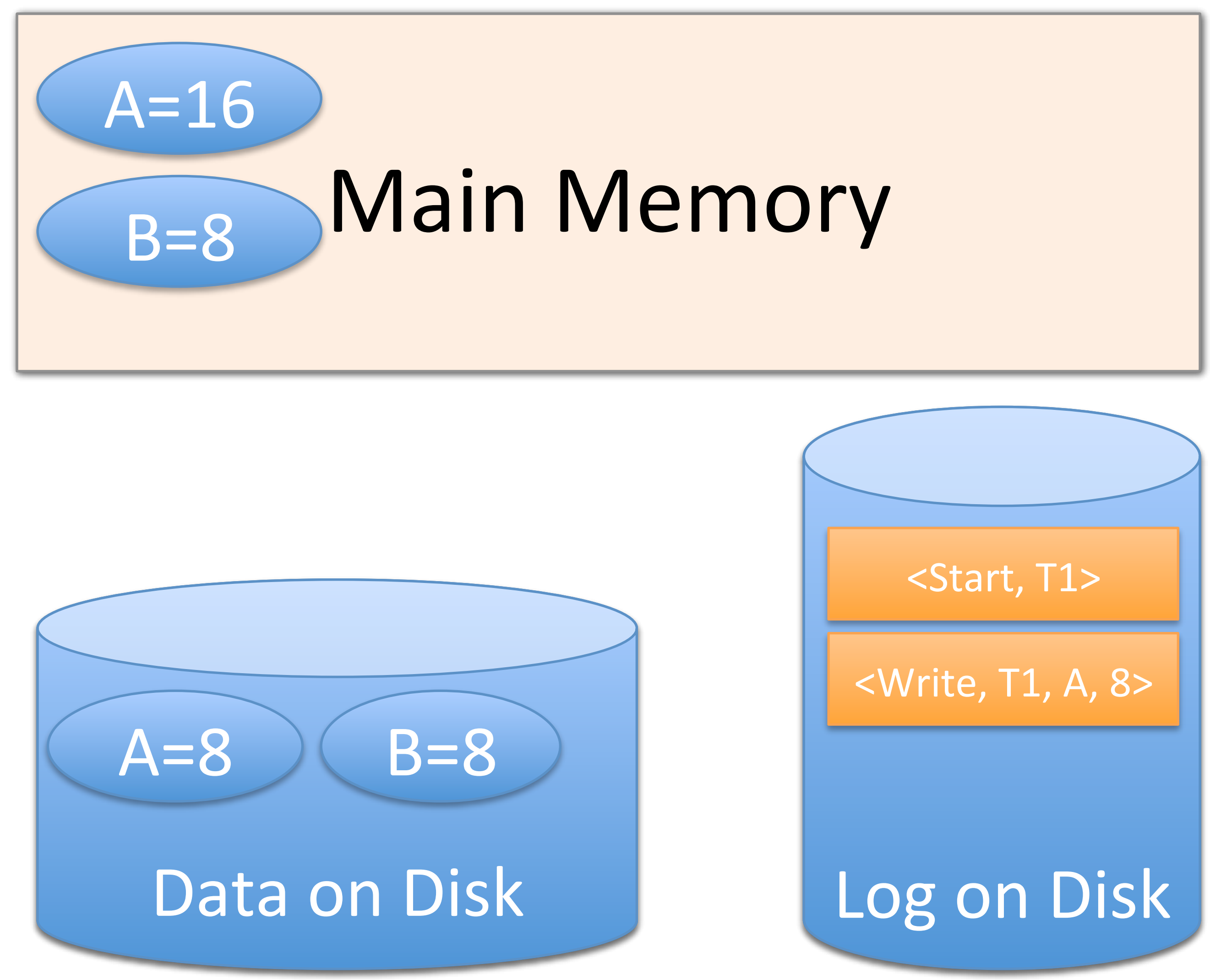
Example: Undo Logging

T1: Read (A, t);
t <— t x 2;
Write(A, t);
Read (B, t);
t <— t x 2;
Write(B, t);



Example: Undo Logging

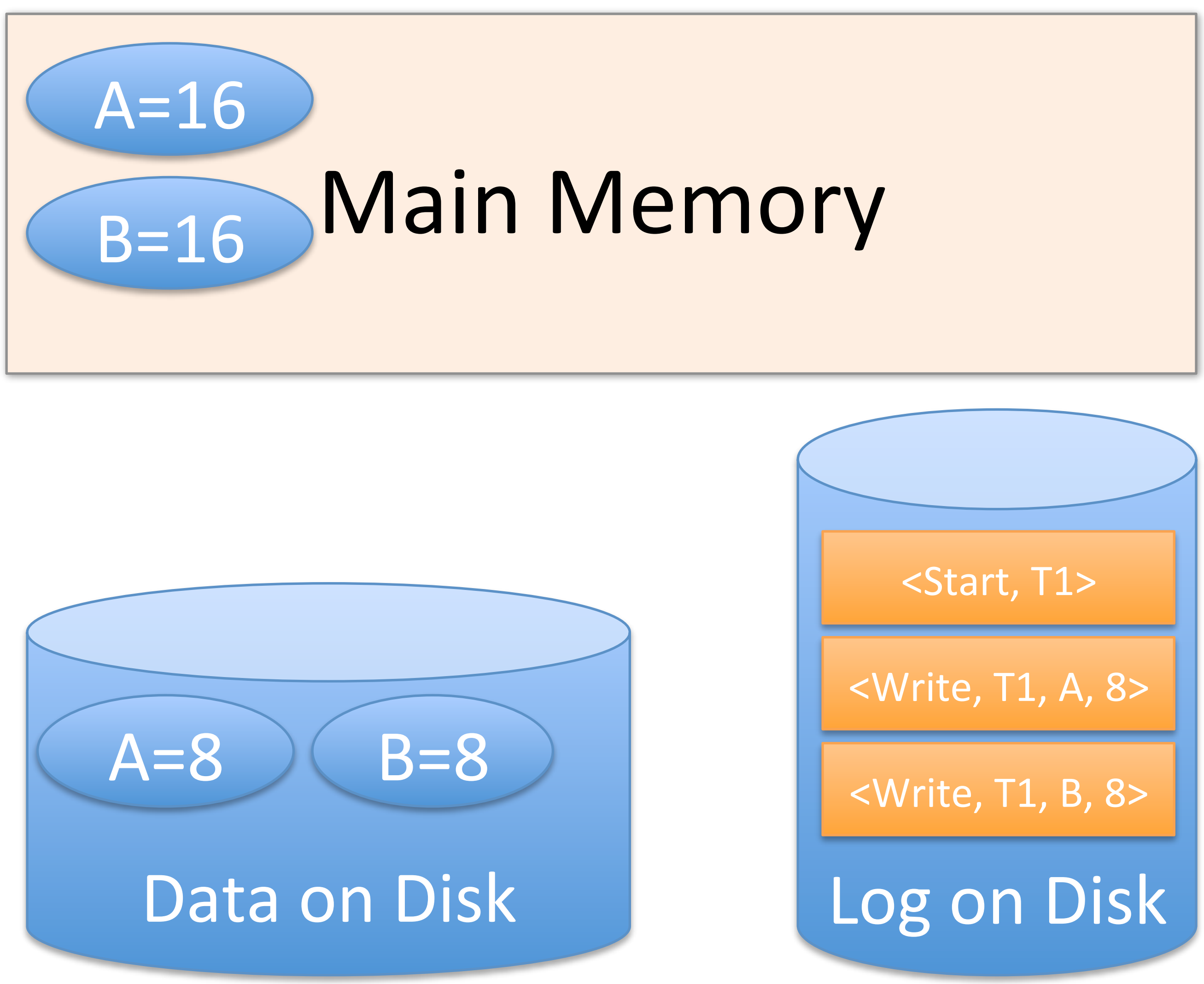
T1: Read (A, t);
t <— t x 2;
Write(A, t);
Read (B, t);
t <— t x 2;
Write(B, t);



If crash occurs now, we can check the log and roll back to the last known state and recover A = 8, B = 8!

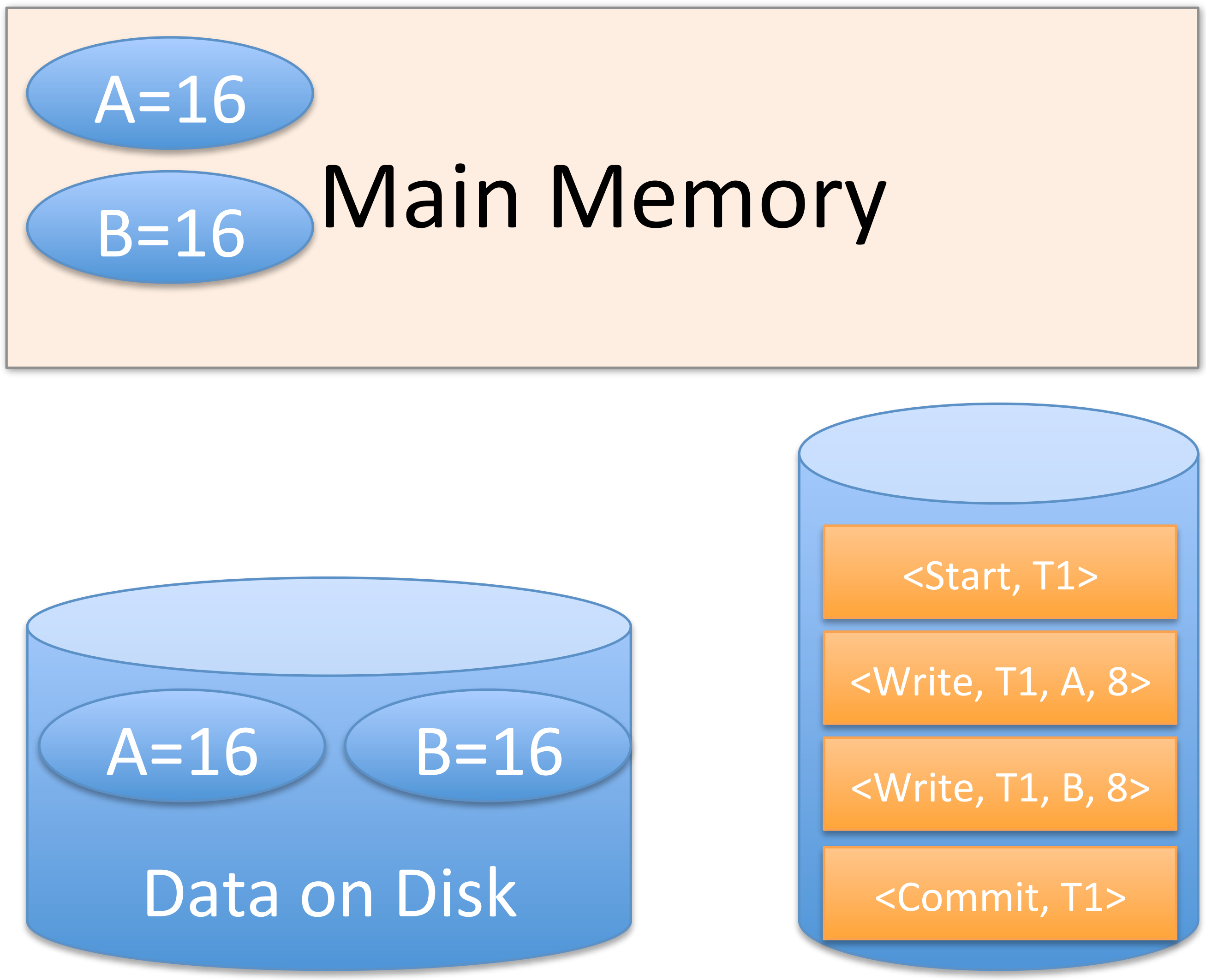
Example: Undo Logging

T1: Read (A, t);
t <— t x 2;
Write(A, t);
Read (B, t);
t <— t x 2;
Write(B, t);



Example: Undo Logging

T1: Read (A, t);
t <— t x 2;
Write(A, t);
Read (B, t);
t <— t x 2;
Write(B, t);



Redo Logging

Idea: save disk I/Os by deferring data changes or do the changes for committed transaction

- New transaction begins — add [start, T] to the log
- Read data — do nothing
- Write data — add [write, T, X, **new_value**], after successful write to log, update X with new value
- Complete transaction — add [commit, T] to log
- Abort transaction — add [abort, T] to log

Checkpoints

- Log grows infinitely — take checkpoints to reduce amount of processing
- Periodically
  - Do not accept new transactions and wait for active ones to finish
  - Write “checkpoint” record to disk
  - Flush all log records and resume transaction processing

http://www.saintlouischeckpoints.com/wp-content/uploads/2013/08/dui20checkpoint200220172011.jpg

Logging Summary

- WAL and recovery protocol are used to
  - Undo actions of aborted transactions
  - Restore the system to a consistent state after a crash
- Helps with atomicity and durability
- But only half the story …

Concurrent Executions

- Multiple transactions should be allowed to run concurrently in the system
    - Increased processor and disk utilization — better transaction throughput
    - Reduced average response time for transactions
- But, interleaving transactions to ensure isolation and handling system crashes are the hard part!

Schedule

- A schedule S of n transactions T1, T2, …, Tn is an ordering of the operations of the transactions
  - For each transaction Ti, the operations in Ti in S must appear in the same order in which they occur in Ti
  - Operations from other transactions Tj can be interleaved with operations of Ti in S
- Schedule represents an actual or potential execution sequence of the transactions

Example: Schedule

Initial DB state: A = 25, B = 25

```
T1: Read(A);
    A <— A+100;
    Write(A);
    Read(B);
    B <— B + 100;
    Write(B);
```

```
T2: Read(A);
    A <— A x 2;
    Write(A);
    Read(B);
    B <— B x 2;
    Write(B);
```

Example: Serial Schedule A

T1	T2	A = 25; B = 25
Read(A); A <— A + 100; Write(A);		A = 125
Read(B); B <— B + 100; Write(B);		B = 125
	Read(A); A <— A x 2; Write(A);	A = 250
	Read(B); B <— B x 2; Write(B);	B = 250

Example: Serial Schedule B

T1	T2	A = 25; B = 25
	Read(A); A <— A x 2; Write(A);	A = 50
	Read(B); B <— B x 2; Write(B);	B = 50
Read(A); A <— A + 100; Write(A);		A = 150
Read(B); B <— B + 100; Write(B);		B = 150

Example: Serial Schedule C

T1	T2
Read(A); A <— A + 100; Write(A);	
	Read(A); A <— A x 2; Write(A);
Read(B); B <— B + 100; Write(B);	
	Read(B); B <— B x 2; Write(B);

A = 25; B = 25

A = 125

A = 250

B = 125

B = 250

Example: Nonserializable Schedule D

T1	T2
Read(A); A <— A + 100; Write(A);	
	Read(A); A <— A x 2; Write(A);
	Read(B); B <— B x 2; Write(B);
Read(B); B <— B + 100; Write(B);	

A = 25; B = 25

A = 125

A = 250

B = 50

B = 150

Serializability

- Want schedules that are “good” regardless of
  - Initial state
  - Transaction semantics
- “Equivalent” to a serial schedule
- Only look at order of read and writes
- Note: if each transaction preserves consistency, every serializable schedule preserves consistency

Conflict

- Pairs of consecutive actions such that if their order is interchanged, the behavior of at least one of the transactions can change
  - Involve the same database element
  - At least one write
- Three types of conflict: read-write conflicts (RW), write-read conflicts (WR), write-write conflicts (WW)

Example: Read-Write Conflict

	T1	T2	
A = 10	BEGIN Read(A);		
		BEGIN Read(A); A <— A * 2; Write(A); COMMIT;	A = 10 A = 20
A = 20	Read(A); COMMIT		

"Unrepeatable read"
- T1 gets different / inconsistent values!

Example: Write-Read Conflict

	T1	T2	
A = 10 A = 12	BEGIN Read(A); A <— A + 2; Write(A);		
		BEGIN Read(A); A <— A * 2; Write(A); COMMIT;	A = 12 A = 24
	Read(B); B <— B + 100; ABORT		

A "dirty read" (reading uncommitted data) means T2's result is based on obsolete / inconsistent value!

Example: Write-Write Conflict

	T1	T2	
A = 10	BEGIN Write(A);		
		BEGIN Write(A); Write(B); COMMIT;	A = 20 B = 100
B = 20	Write(B); COMMIT		

Overwriting uncommitted data results in partially-lost update and not equivalent to any serial schedule

Serializability Definitions

- S1, S2 are **conflict equivalent** schedules if S1 can be transformed into S2 by a series of swaps on non-conflicting actions

- A schedule is **conflict serializable** if it is conflict equivalent to some serial schedule

    - Maintains consistency & isolation!

Example: Not conflict serializable

T1	T2
BEGIN	
Read(A); Write(A);	
	BEGIN Read(A); Write(A); Read(B); Write(B); COMMIT;
Read(B); Write(B);	
COMMIT	

Conflict 1

Conflict 2

Both conflicts will not happen in this order for a serial schedule!

Example: Serializable vs Conflict Serializable

T1	T2	T3
BEGIN Read(A);		
	BEGIN Write(A); COMMIT	
Write(A) COMMIT		
		BEGIN Write(A); COMMIT

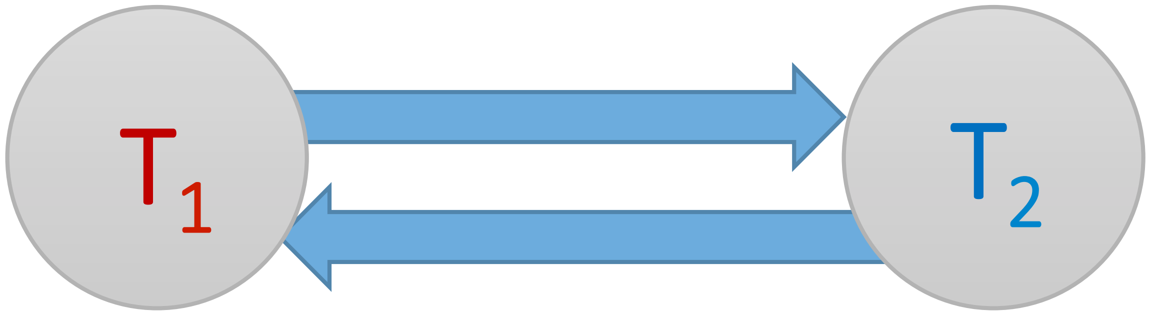
- Equivalent to T1, T2, T3, so serializable
- Not conflict equivalent to T1, T2, T3 so not conflict serializable
- Conflict serializable => serializable but not the other way around!

Precedence (Serialization) Graph

- Graph with directed edges
  - Nodes are transactions in S
  - Edge is created from Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj
- Schedule is serializable if and only if precedence graph has no cycles!

Example: Precedence Graph

T1	T2
Read(A); A <— A + 100; Write(A);	
	Read(A); A <— A x 2; Write(A);
	Read(B); B <— B x 2; Write(B);
Read(B); B <— B + 100; Write(B);	



A non-conflict serializable schedule has a cycle!

Locks: Basic Idea

- Each time you want to read/write an object, obtain a lock to secure permission to read/write object
- When completed, unlock removes permissions from data item
- Ensure transactions remain isolated and follow serializable schedules

T1	T2
BEGIN Lock(A) Read(A);	
	BEGIN Lock(A)
Write(A) Unlock(A) COMMIT	denied since T1 has lock
	Read(A) Write(A) Unlock(A) COMMIT

Basic Locking

- Two lock modes: shared (read), exclusive (write)
- If a transaction wants to read an object, it must first request a shared lock on that object
- If a transaction wants to modify an object, it must first request an exclusive lock on that object

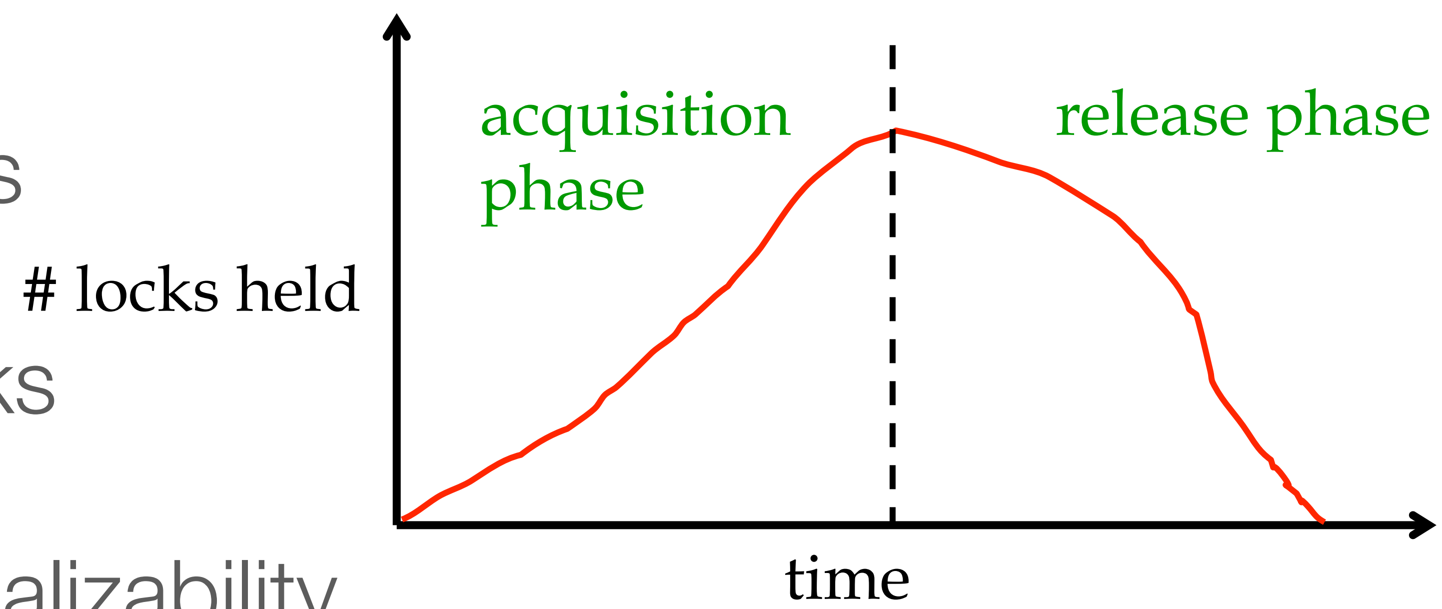
	Shared	Exclusive
Shared	Yes	No
Exclusive	No	No

Example: Basic Locking Insufficient

	T1	T2	
A = B A = 100 A = 105	Exclusive-Lock(A); Read(A); A <— A + 5; Write(A); Unlock(A);		
		Exclusive-Lock(A); Read(A); A <— A x 2; Write(A); Unlock(A);	A = 105 A = 210
		Exclusive-Lock(B); Read(B); B <— B x 2; Write(B); Unlock(B)	B = 100 B = 200
B = 200 B = 205	Exclusive-Lock(B); Read(B); B <— B + 5; Write(B); Unlock(B)	A =/= B => not conflict-serializable!	

Two-phase Locking (2PL)

- All lock requests precede all unlock requests
    - Phase 1: obtain locks
    - Phase 2: release locks
- Guarantees conflict serializability
- Does not prevent cascading aborts (where aborting one transaction causes one or more other transactions to abort)



Example: Cascading Abort

T1	T2
Exclusive-Lock(A); Read(A); A <— A + 5; Write(A); Exclusive-Lock(B) Unlock(A);	
	Exclusive-Lock(A); Read(A); A <— A x 2; Write(A); Exclusive-Lock(B); Unlock(A);
	Read(B); B <— B x 2; Write(B); Unlock(B)
Read(B); B <— B + 5; Write(B); Unlock(B)	

cannot obtain lock on B until T1 unlocks

But what if we abort here?

Strict Two-phase Locking (Strict 2PL)

- Only release locks at commit / abort time
  - A transaction that writes will block all other readers until the transaction commits or aborts
- Used in many commercial DBMS systems
  - Oracle is notable exception
- Downside: not deadlock free

Example: Deadlock

T1	T2
Shared-Lock(Y); Read(Y);	
	Shared-Lock(X); Read(X);
Exclusive-Lock(X); Write(X);	
	Exclusive-Lock(Y);

T1 and T2 follow the strict 2PL policy but are deadlocked!

Deadlock Protocols

Different ways to deal with deadlock

- Deadlock prevention
  - Rigorous locking protocol — acquire all locks in advance
  - Timeout — waits some amount of time then roll back
- Deadlock detection
  - Construct and maintain graph

Transactions & Concurrency: Recap

- ACID
- Logging
  - WAL
  - Checkpoints
- Conflict Serializable Schedules
  - Locking: Basic, 2PL, Strict 2PL
  - Deadlock