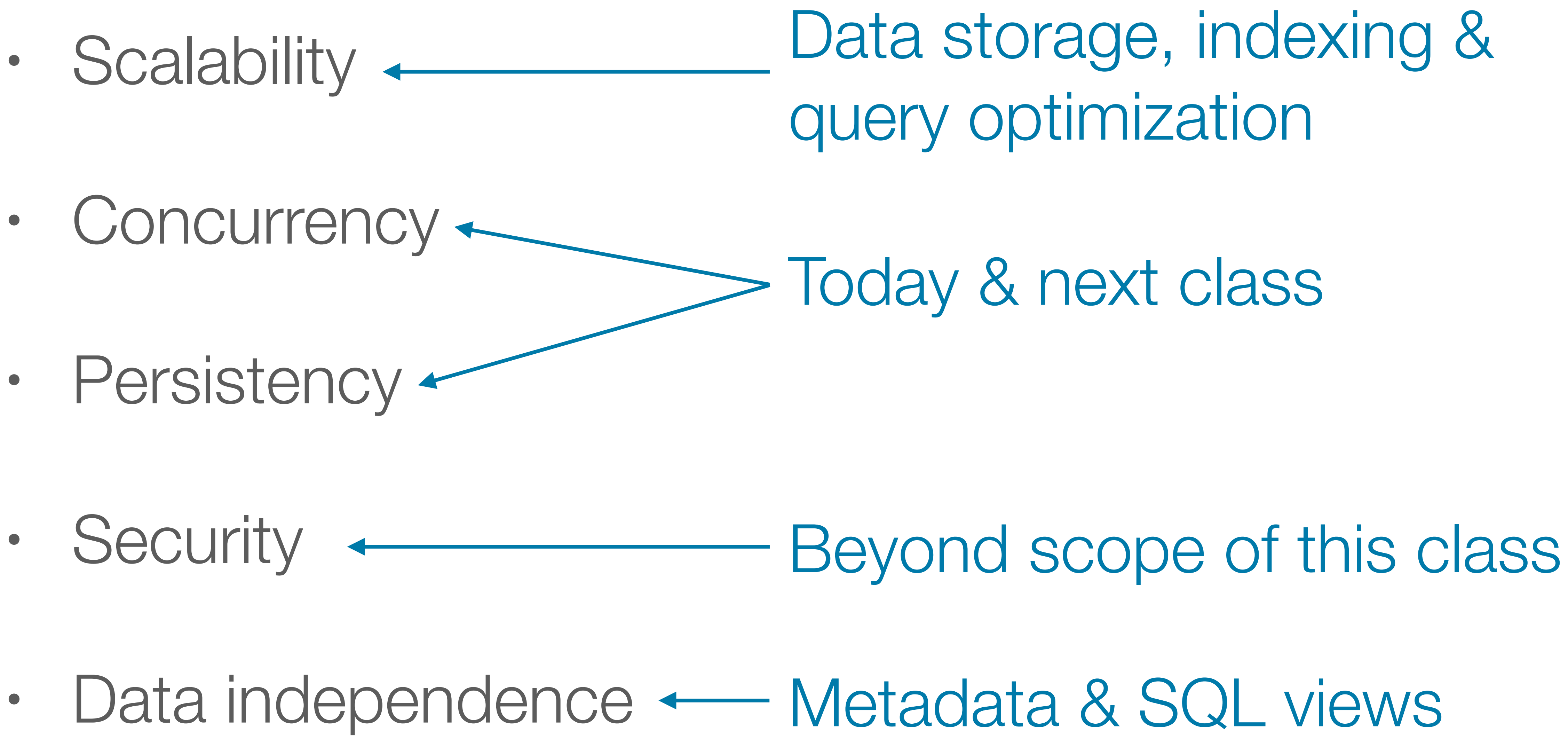
Transaction Management & Concurrency Control

CS 377: Database Systems

Review: Database Properties



- Scalability
- Concurrency
- Persistency
- Security
- Data independence

Data storage, indexing & query optimization

Today & next class

Beyond scope of this class

Metadata & SQL views

Review: Disk vs Main Memory

- Disk
    - Slow — sequential access
    - Durable — once on disk, data is safe
    - Cheap

- Main memory (RAM)
    - Fast
    - Volatile — data can be lost
    - Expensive
    -

Memory Model

1. Local: Each process in a DBMS has its own local memory, where it stores values that only it “sees”
2. Global: Each process can read/write to/from shared data in main memory
3. Disk: Global memory can be read from / write to disk

	Local	Global
Main Memory (RAM)	1	2
Disk		3

How do we effectively utilize both to ensure certain guarantees?

Transaction: Motivation

- 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);
```

Database crash! What happens?

What if wife also withdraws money before the money is deducted?

Transaction: Motivation

- 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: Basic Definition

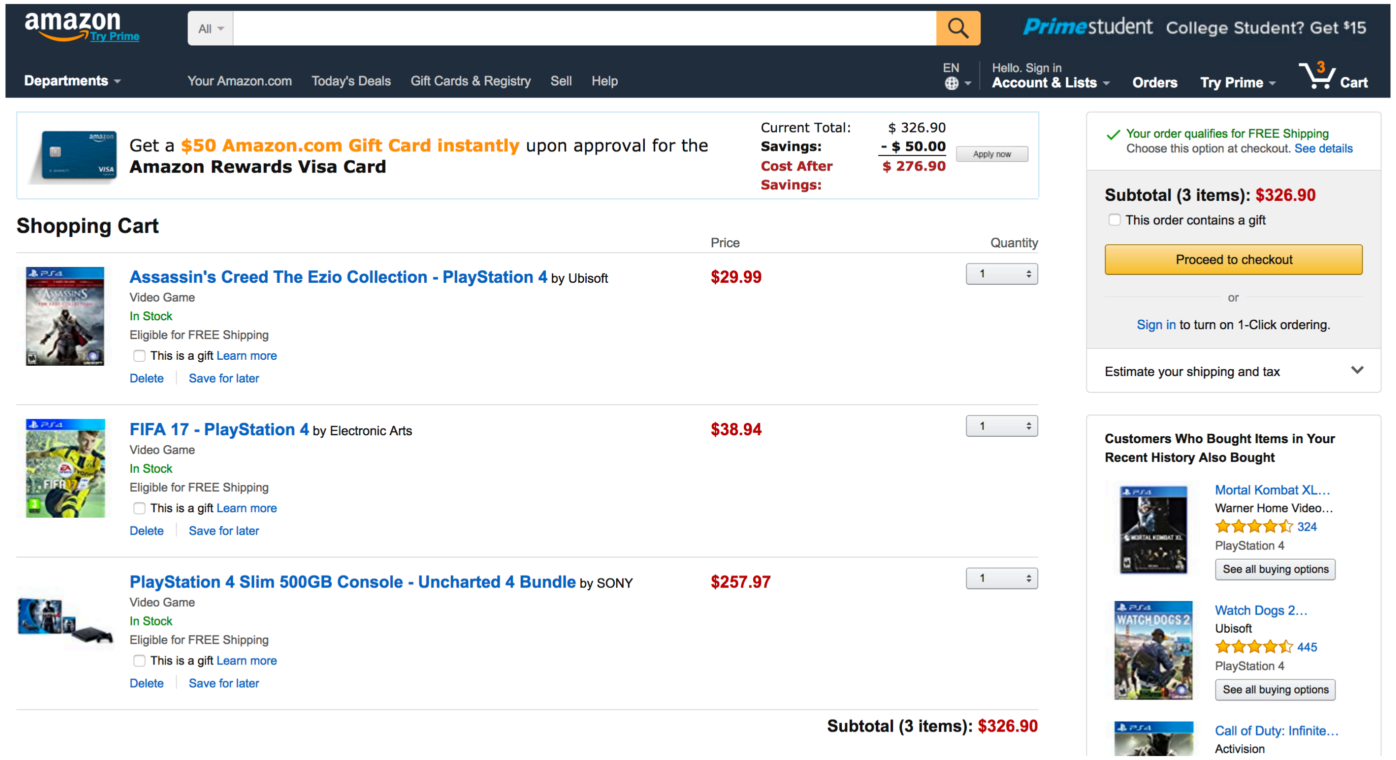
- A transaction (TXN) is a sequence of one or more operations (reads or writes) which reflects a single real-world transition

- TXN is a collection of operations that form a single atomic logical unit of execution

- TXNs must leave the database in a consistent state — it either happened completely or not at all

Transaction: Example

- Transfer money between accounts
- Purchase a group of products
- Register for a class (waitlist or signed up)



Get a **$50 Amazon.com Gift Card instantly** upon approval for the **Amazon Rewards Visa Card**

Current Total:	$ 326.90
Savings:	**- $ 50.00**
Cost After Savings:	**$ 276.90**

Shopping Cart

	Price	Quantity
Assassin's Creed The Ezio Collection - PlayStation 4 by Ubisoft Video Game In Stock Eligible for FREE Shipping ☐ This is a gift Learn more Delete | Save for later	**$29.99**	1
FIFA 17 - PlayStation 4 by Electronic Arts Video Game In Stock Eligible for FREE Shipping ☐ This is a gift Learn more Delete | Save for later	**$38.94**	1
PlayStation 4 Slim 500GB Console - Uncharted 4 Bundle by SONY Video Game In Stock Eligible for FREE Shipping ☐ This is a gift Learn more Delete | Save for later	**$257.97**	1

Subtotal (3 items): $326.90

✓ Your order qualifies for FREE Shipping
Choose this option at checkout. See details

Subtotal (3 items): $326.90

☐ This order contains a gift

or

Sign in to turn on 1-Click ordering.

Estimate your shipping and tax

Customers Who Bought Items in Your Recent History Also Bought

Mortal Kombat XL…
Warner Home Video…
★★★★☆ 324
PlayStation 4

Watch Dogs 2…
Ubisoft
★★★★☆ 445
PlayStation 4

Call of Duty: Infinite…
Activision

Transaction: Operations

- For purpose of class, assume only two operations
  - READ(X) - retrieval
  - WRITE(X) - insert, delete, update
- In reality — users can do much more and databases have more to deal with

Transaction: ACID

A

Atomicity: Transactions are all or nothing

C

Consistency: Only valid data is saved

I

Isolation: Transactions do not affect each other

D

Durability: Written data will not be lost

Transaction: ACID

- 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.

Transaction: ACID

- 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: ACID Challenges

- Need to handle failures (e.g., power outages, bad network connection)
- Users may abort the program: need to “rollback the changes”
- Many users executing concurrently
- Maintain ACID with performance!

Transaction: Is ACID Good?

- Extremely important and successful paradigm
- Many debates over ACID — both historically and currently
- Many newer "NoSQL" DBMS relax ACID (more on this later)

Transaction: Management

- 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
- Rely on 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

Transaction: SQL

- “Ad-hoc” SQL: Each statement = one transaction
- Multiple statements can be grouped together as a transaction
- Example: Transfer money between two accounts

```
START TRANSACTION
     UPDATE Account SET amount = amount – 100
     WHERE name = ‘Bob’
     UPDATE Account SET amount = amount + 100
     WHERE name = ‘Alice’
COMMIT
```

Transaction: SQL

- 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

- Essential for reliable DBMS usage
  - DBMS may experience crashes (e.g., power outages, etc.)
  - Individual TXNs may be aborted (e.g., by user)
- How to make sure TXNs are either durably stored in full or not at all?

Recovery: Protection

```sql
INSERT INTO SmallProduct(name, price)
    SELECT pname, price
    FROM Product
    WHERE price <= 0.99
```

```sql
DELETE Product
    WHERE price <=0.99
```

Crash
or
Abort

What goes wrong?

Recovery: Protection

```
START TRANSACTION
        INSERT INTO SmallProduct(name, price)
            SELECT pname, price
            FROM Product
            WHERE price <= 0.99

        DELETE Product
            WHERE price <=0.99
    COMMIT OR ROLLBACK
```

Now we’re okay — how do we achieve this?

Recovery: System Log

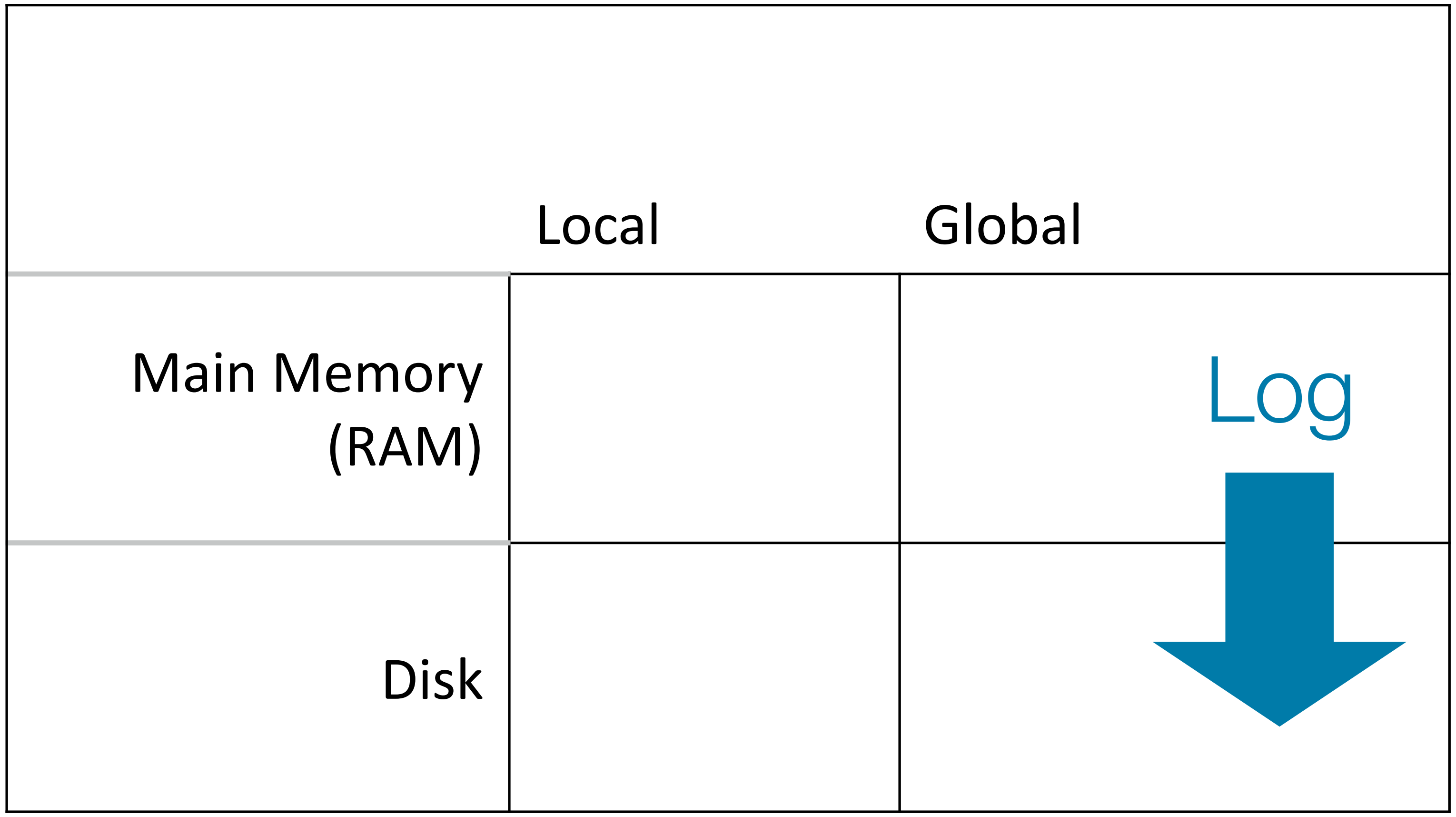
Idea: Keep a system log and perform recovering when necessary

- 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

Log: 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

Log: Memory Model



	Local	Global
Main Memory (RAM)		Log
Disk		

Assume log is on stable disk storage — spans both main memory and disk and every so often will “flush” (write) to disk

Log: Why Bother?

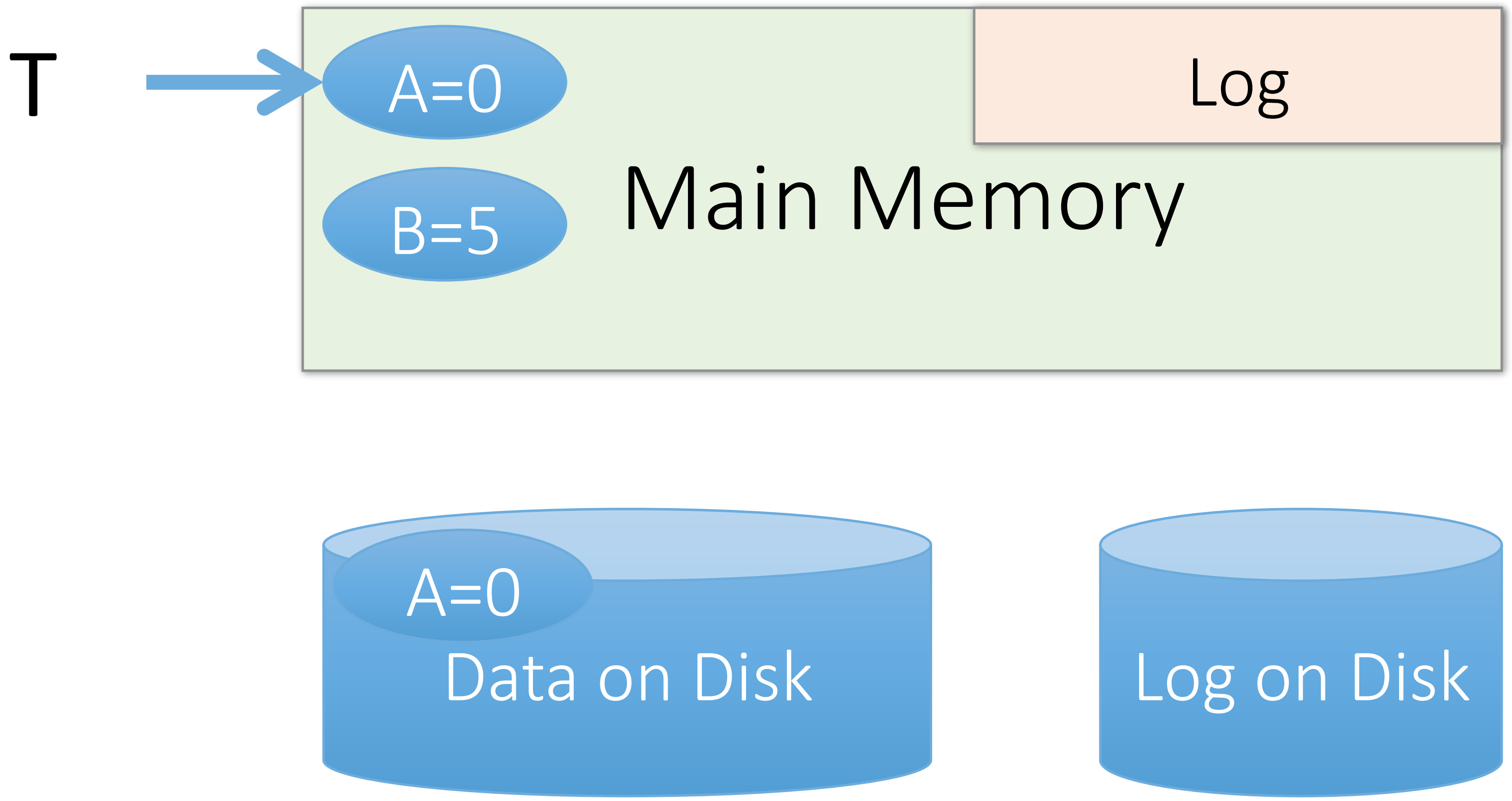
- Can’t we just write transaction to disk only once whole transaction is completed?
    - With unlimited memory and time, this could work…
- What if there isn’t enough space for a full transaction?
- What if one transaction takes very long?

Write Ahead Logging (WAL)

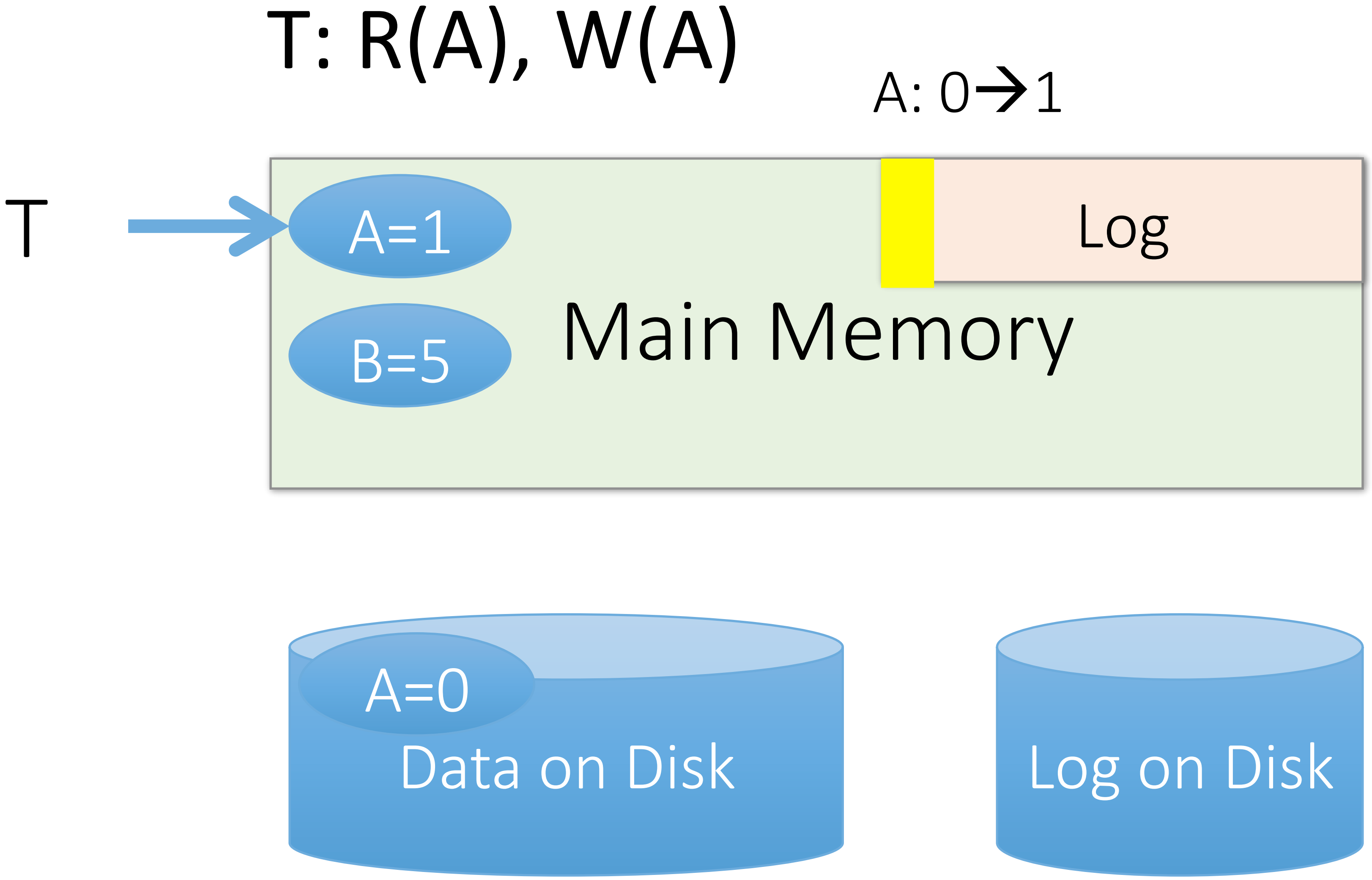
- All modifications are written to a log before they are applied to database
    - Each update is logged before the corresponding data page goes to storage —> atomicity
    - Must write all log records for a TXN before commit —> durability

WAL: Pictorially

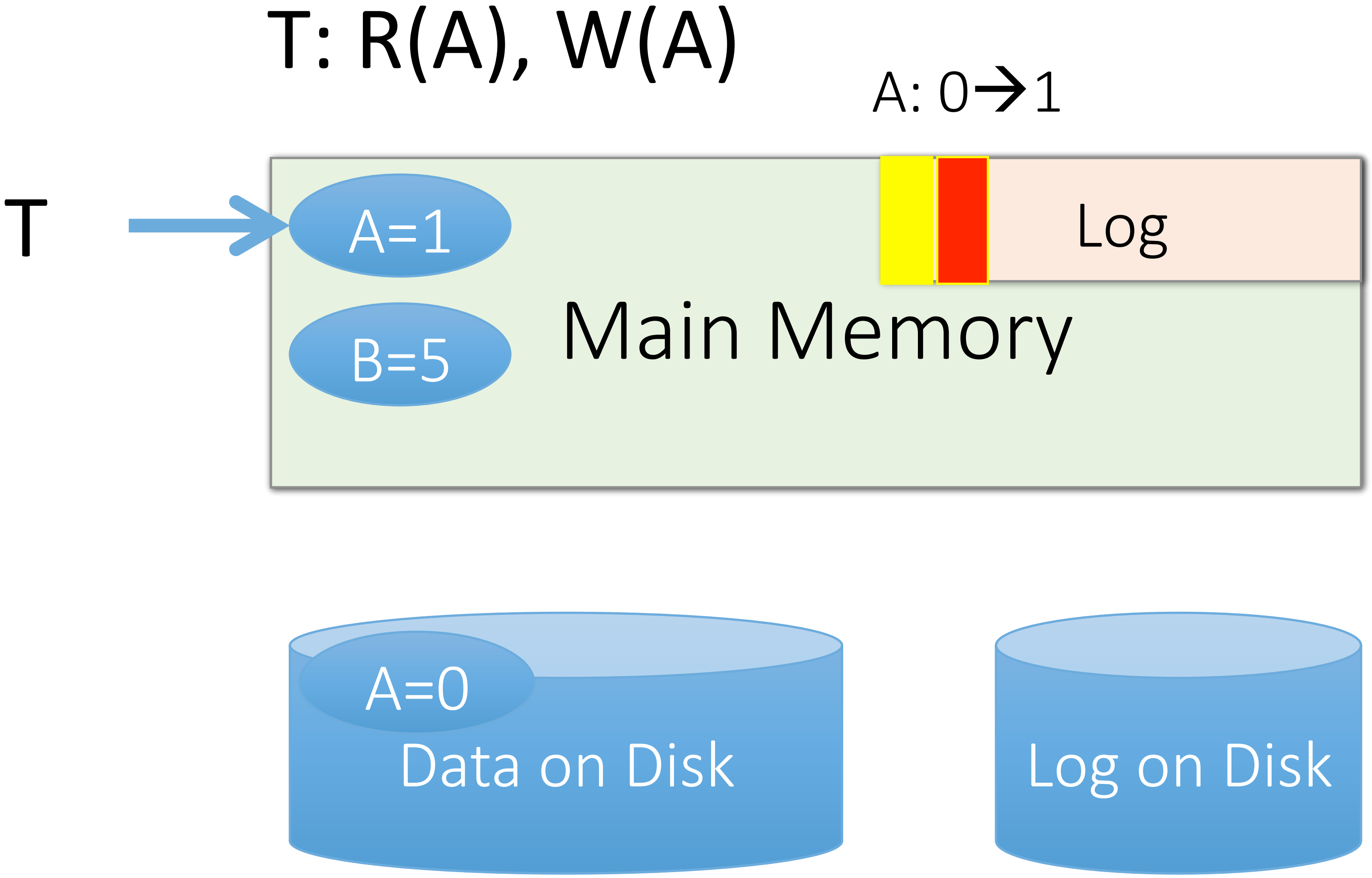
T: R(A), W(A)



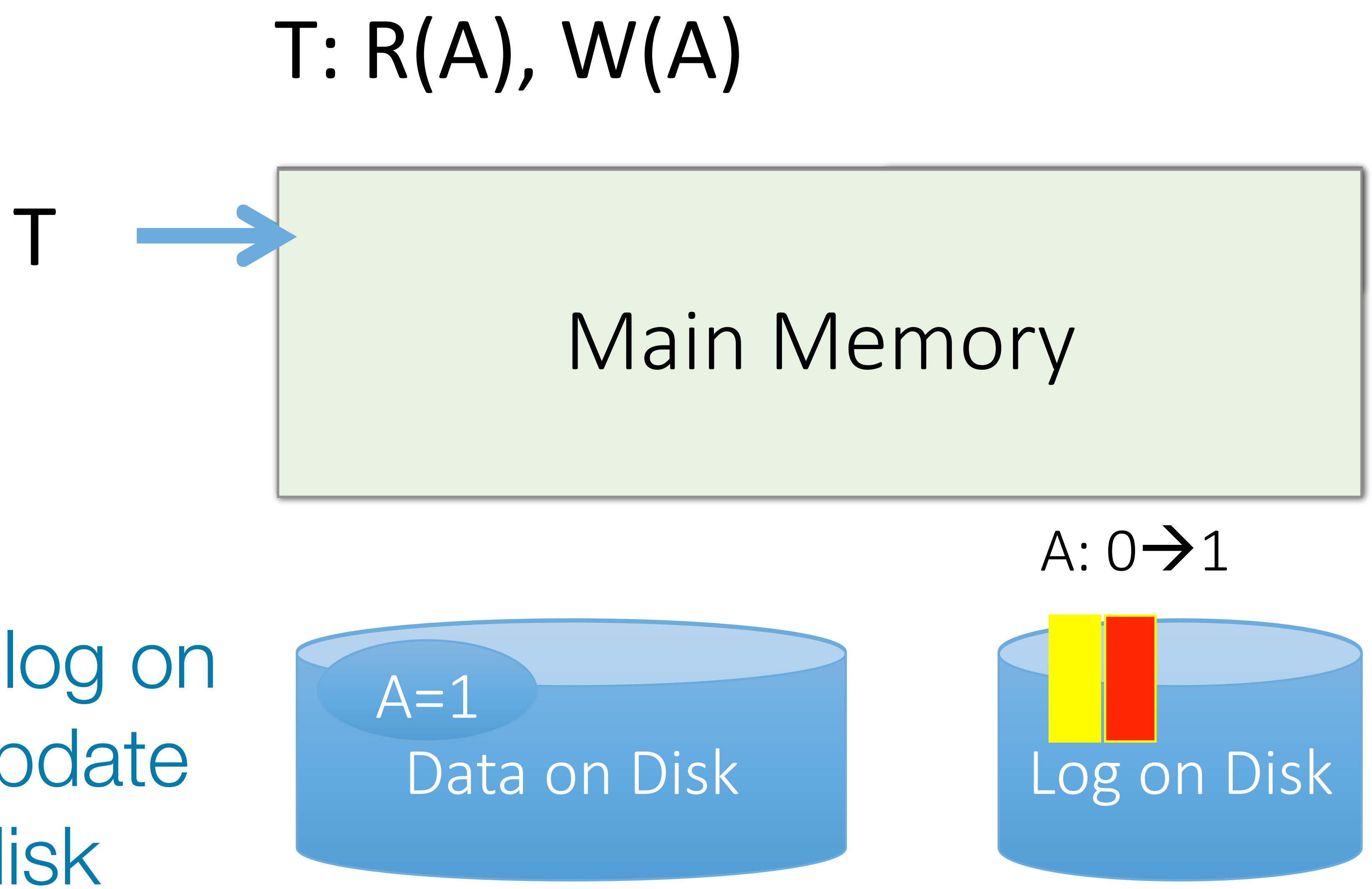
WAL: Pictorially



WAL: Pictorially



WAL: Pictorially



T: R(A), W(A)

T

Main Memory

A: 0→1

A=1

Data on Disk

Log on Disk

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

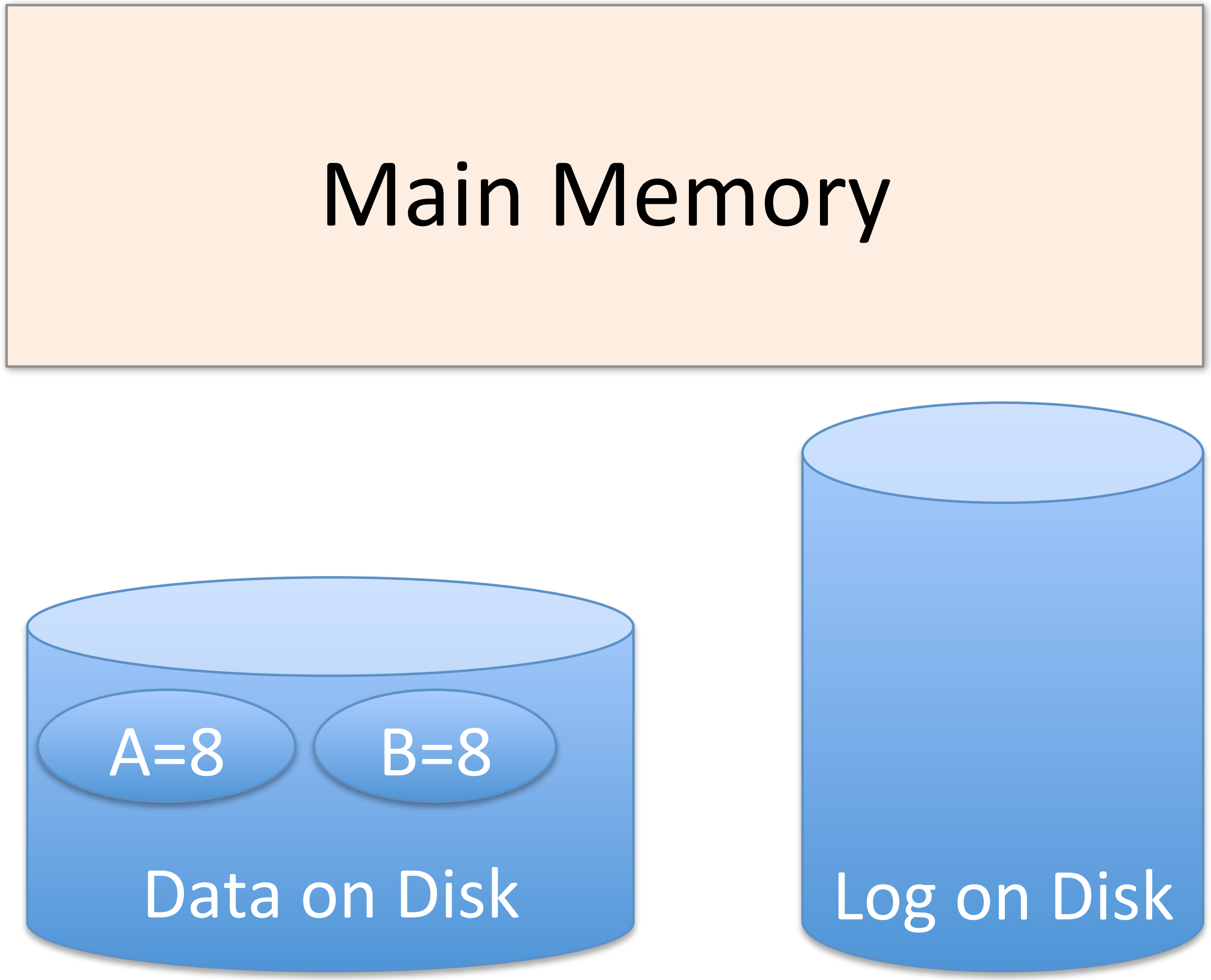
Undo Logging

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

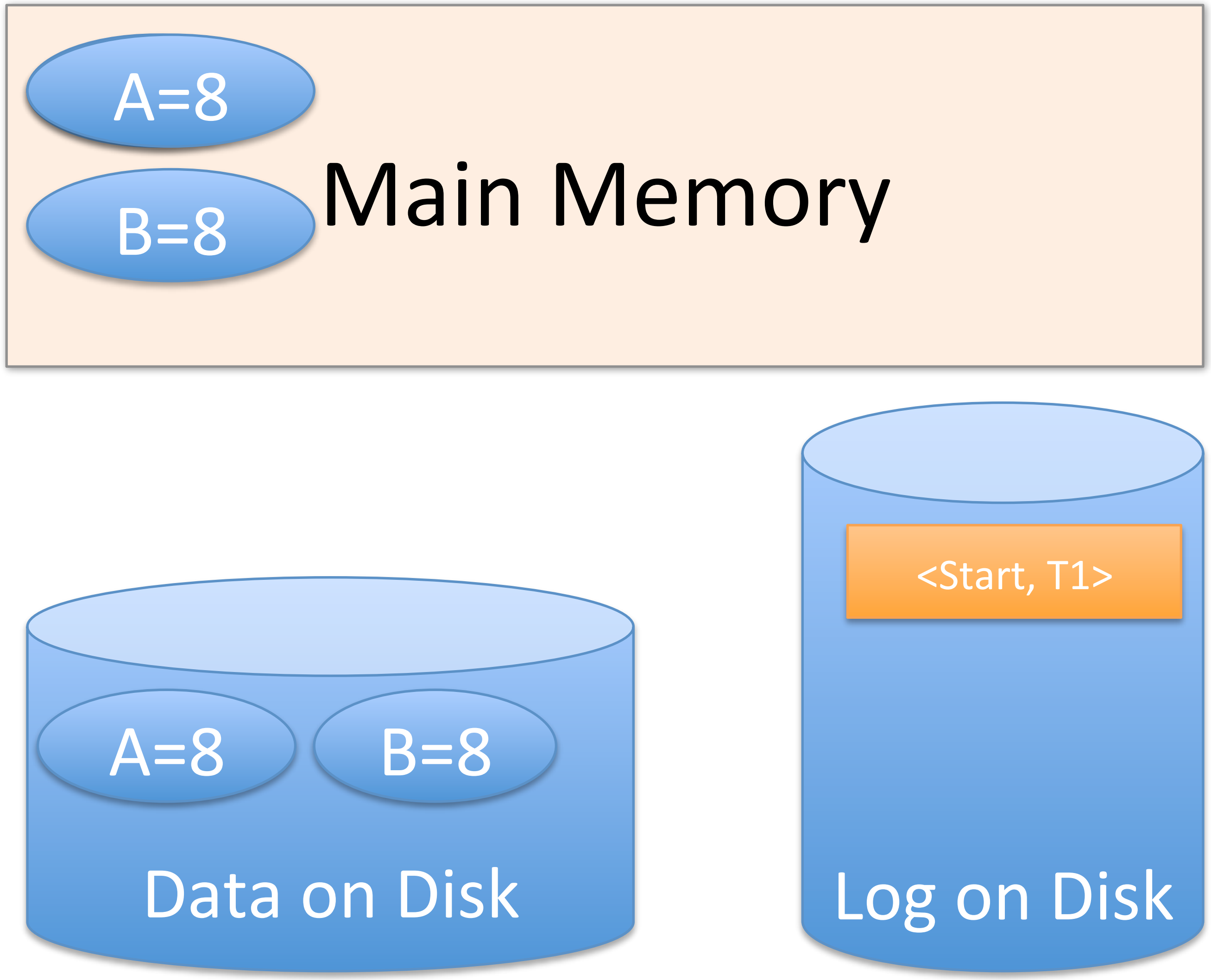
Example: Undo Logging

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



Example: Undo Logging

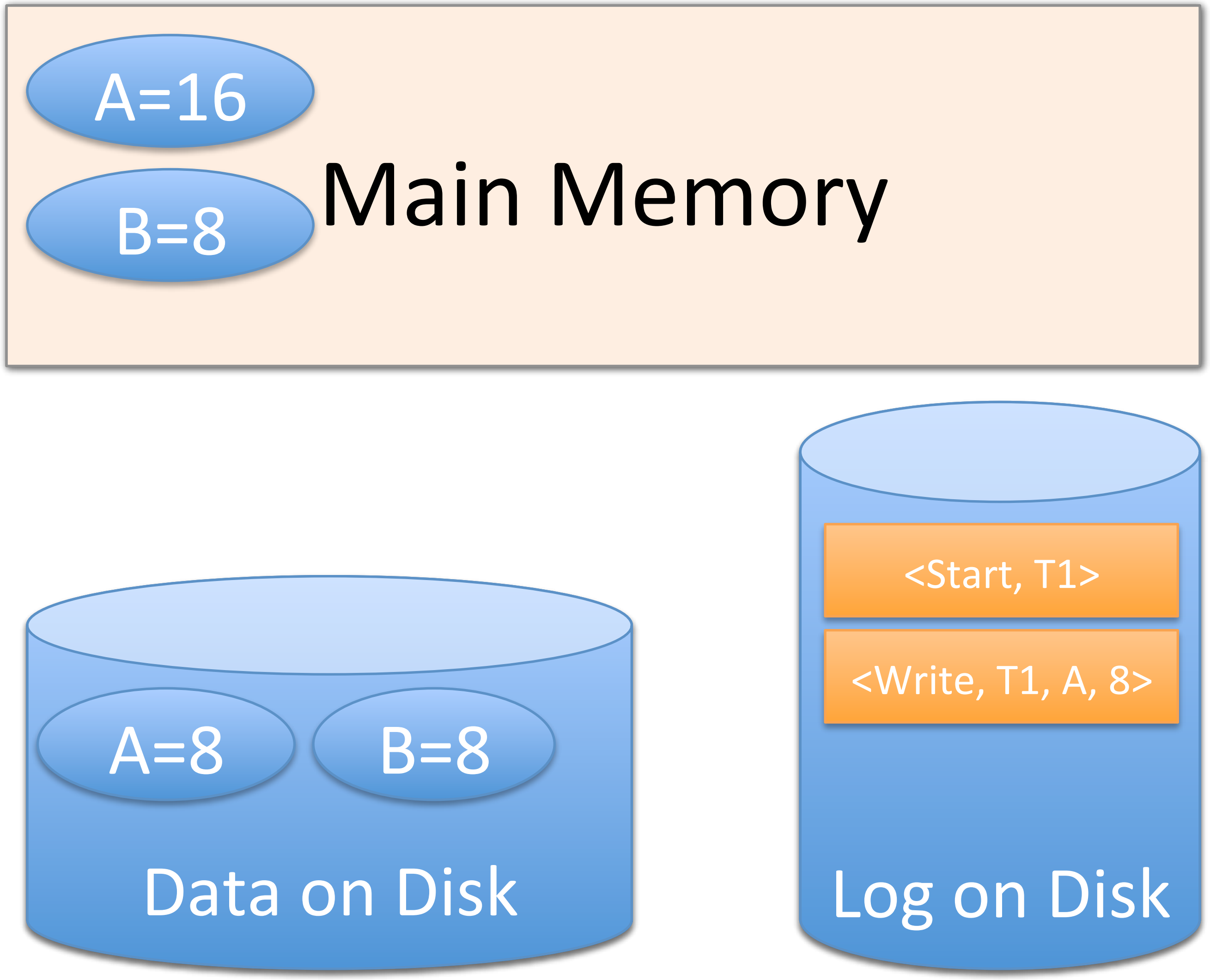
```
T1:   Read (A, t);
      t <— t x 2;
      Write(A, t);
      Read (B, t);
      t <— t x 2;
      Write(B, t);
```



Example: Undo Logging

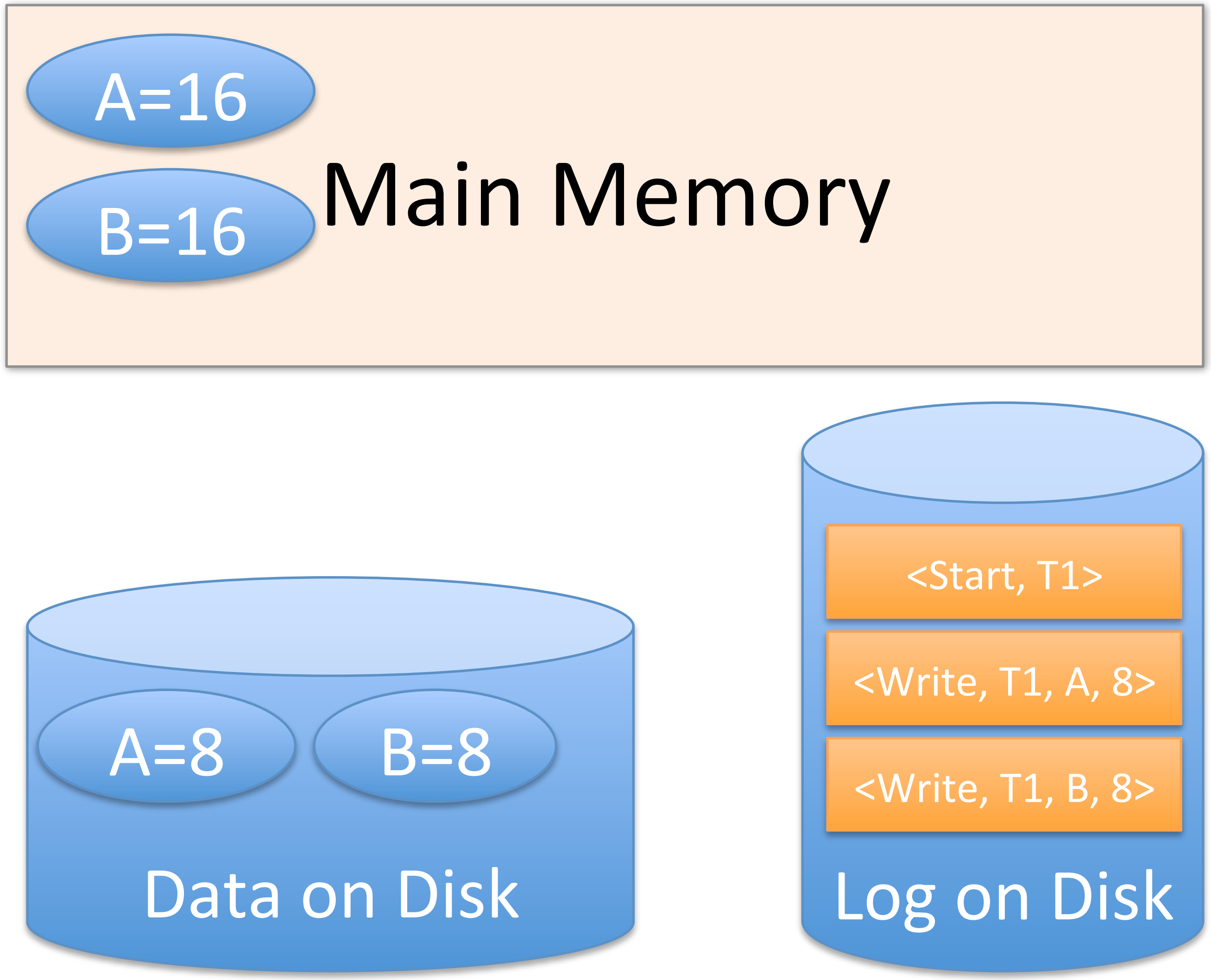
```
T1:    Read (A, t);
       t <— t x 2;
       Write(A, t);
       Read (B, t);
       t <— t x 2;
       Write(B, t);
```

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



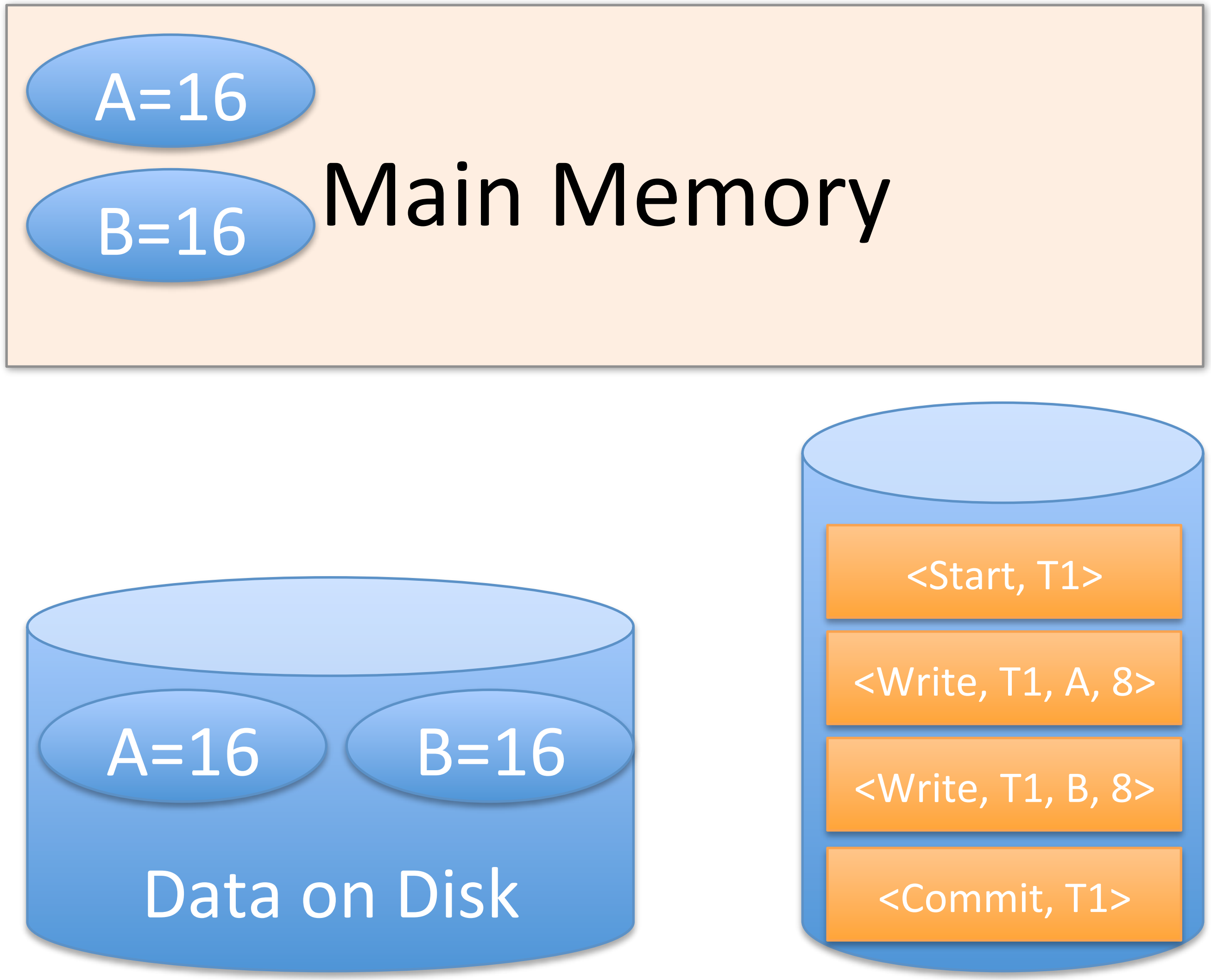
Example: Undo Logging

```
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);
```



A=16

B=16

Main Memory

A=16

B=16

Data on Disk

<Start, T1>

<Write, T1, A, 8>

<Write, T1, B, 8>

<Commit, T1>

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!

Example: Concurrent Executions

```
T1: START TRANSACTION
      UPDATE Accounts
      SET Amt = Amt + 100
      WHERE Name = ‘Alice’

      UPDATE Accounts
      SET Amt = Amt - 100
      WHERE Name = ‘Bob’
COMMIT
```

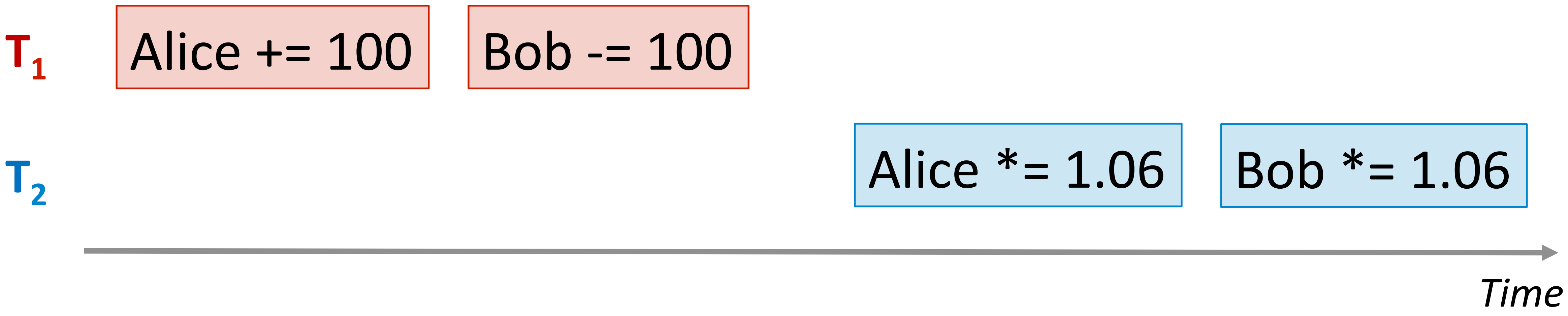
```
T2: START TRANSACTION
      UPDATE Accounts
      SET Amt = Amt * 1.06
COMMIT
```

Transaction 1: Bob transfers money to Alice

Transaction 2: Bank pays interest for all accounts

Example: Serial Executions

Scenario 1:

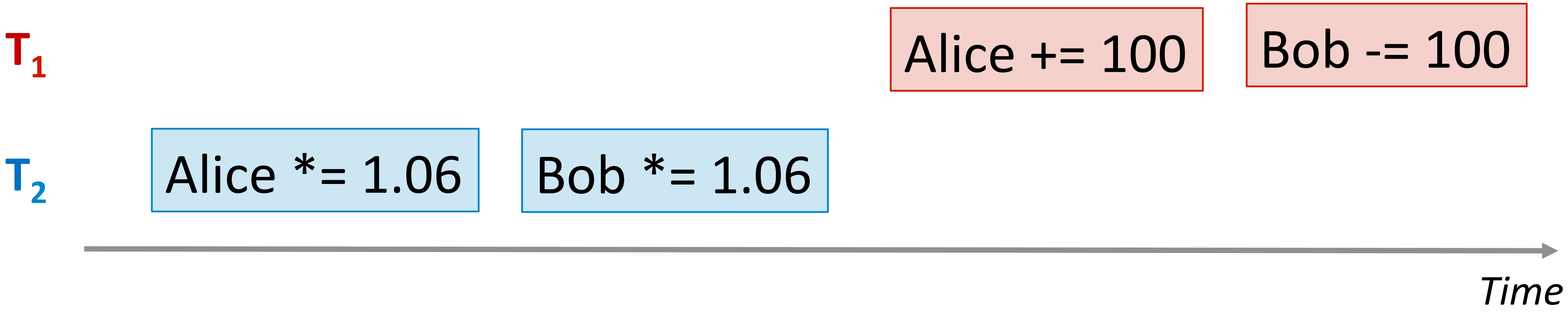


T_1 Alice += 100 Bob -= 100

T_2 Alice *= 1.06 Bob *= 1.06

Time

Scenario 2:



T_1 Alice += 100 Bob -= 100

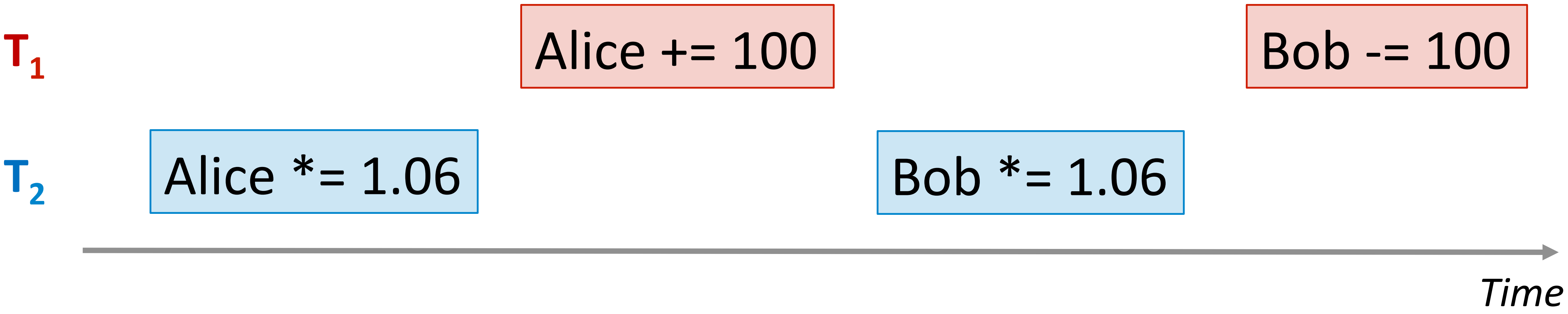
T_2 Alice *= 1.06 Bob *= 1.06

Time

Either scenario could occur in DBMS

Example: Concurrent Executions

Scenario 3: Interleave TXNs



T_1 Alice += 100 Bob -= 100

T_2 Alice *= 1.06 Bob *= 1.06

Time

Is this okay? Does the result look like what would occur if we only ran in serial?

Interleaving Transactions

- Why bother? Interleaving might lead to anomalous outcomes
  - Individual transactions might be slow — should other users wait for this one transaction to finish?
  - Disk access may be slow — let some TXNs use CPUs while others access disk
  - This can lead to large differences in database performance

Schedule

- A schedule S of n transactions T_1, T_2, …, T_n is an ordering of the operations of the transactions
    - For each transaction T_i, the operations in T_i in S must appear in the same order in which they occur in T_i
    - Operations from other transactions T_j can be interleaved with operations of T_i 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

T_1	T_2
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

T_1	T_2	
	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: Interleaved Schedule C

T_1	T_2
Read(A); A <— A + 100; A = 125 Write(A);	
	Read(A); A <— A x 2; A = 250 Write(A);
Read(B); B <— B + 100; B = 125 Write(B);	
	Read(B); B <— B x 2; B = 250 Write(B);

Same result as if I ran T_1 first then T_2!

Example: Interleaved Schedule D

T_1	T_2
Read(A); A <— A + 100; Write(A); A = 125	
	Read(A); A <— A x 2; Write(A); A = 250
	Read(B); B <— B x 2; Write(B); B = 50
Read(B); B <— B + 100; Write(B); B = 150	

Different than running in serial — not serializable

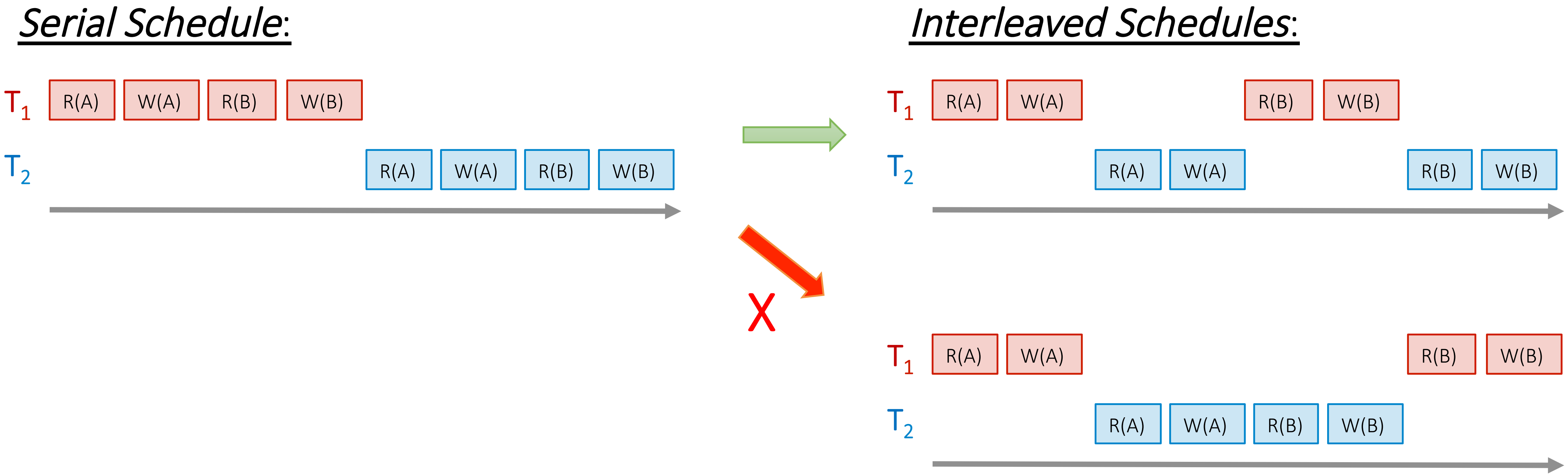
Serializability

- 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

Interleaving TXNs: What goes wrong?

- Various anomalies which break isolation / serializability
- Occur because of / with certain “conflicts” between interleaved transaction
- Note that conflicts can occur without causing anomalies

Schedules: “Good” vs “Bad”



***Serial Schedule*:**

T$_1$ R(A) W(A) R(B) W(B)

T$_2$ R(A) W(A) R(B) W(B)

***Interleaved Schedules*:**

T$_1$ R(A) W(A) R(B) W(B)

T$_2$ R(A) W(A) R(B) W(B)

X

T$_1$ R(A) W(A) R(B) W(B)

T$_2$ R(A) W(A) R(B) W(B)

Want to develop ways to determine “good” vs “bad” schedules

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

T_1	T_2
BEGIN Read(A);	
	BEGIN Read(A); A <— A * 2; Write(A); COMMIT;
Read(A); COMMIT	

A = 10

“Unrepeatable read”
- T1 gets different / inconsistent values!

A = 10

A = 20

A = 20

Example: Write-Read Conflict

T_1	T_2
BEGIN Read(A); A <— A + 2; Write(A);	
	BEGIN Read(A); A <— A * 2; Write(A); COMMIT;
Read(B); B <— B + 100; ABORT	

A = 10
A = 12

A = 12
A = 24

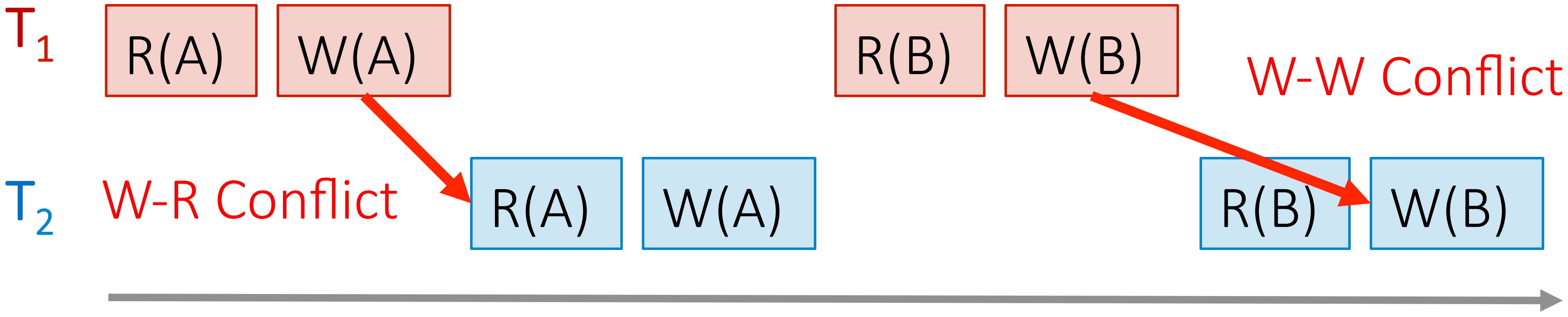
A "dirty read" (reading uncommitted data) means T2's result is based on obsolete / inconsistent value!

Example: Write-Write Conflict

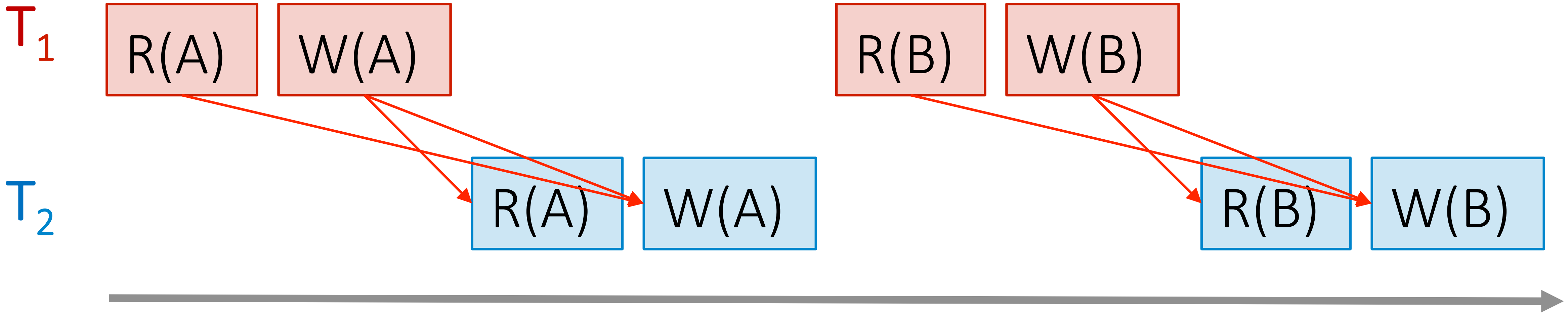
T_1	T_2
BEGIN A = 10 Write(A);	
	BEGIN Write(A); A = 20 Write(B); B = 100 COMMIT;
B = 20 Write(B); COMMIT	

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

Conflict: Example



Conflict: Example

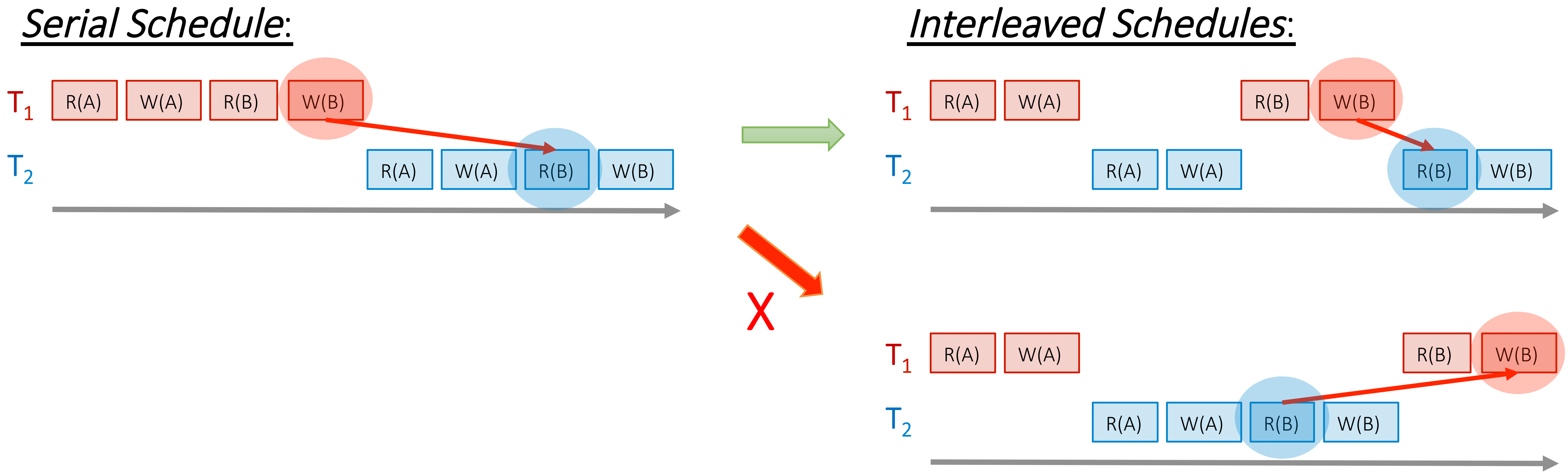


All "conflicts"!

Serializability Definitions

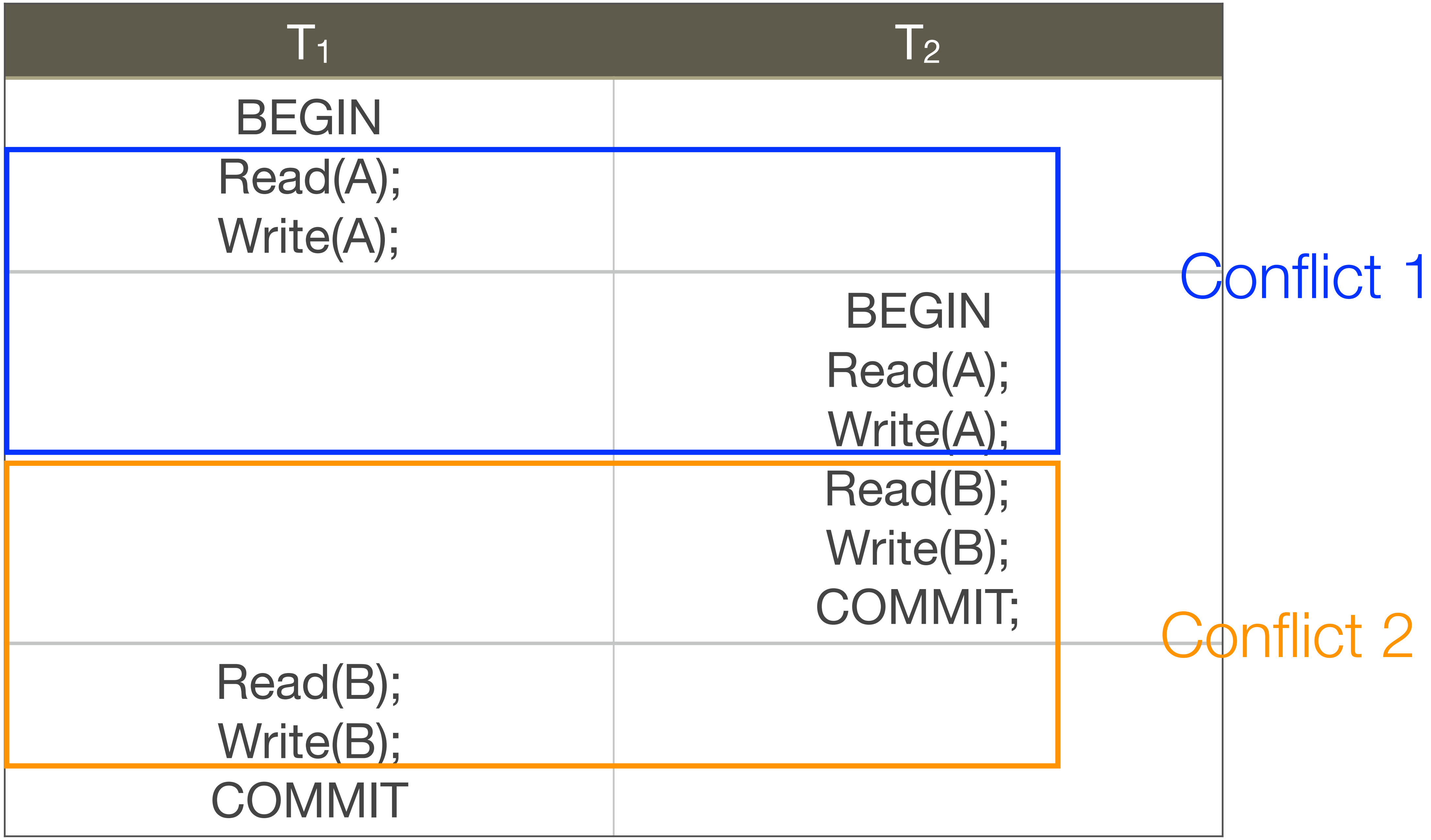
- S_1, S_2 are **conflict equivalent** schedules if S_1 can be transformed into S_2 by a series of swaps on non-conflicting actions
  - Every pair of conflicting actions of two TXNs are ordered the same way
- A schedule is **conflict serializable** if it is conflict equivalent to some serial schedule
  - Maintains consistency & isolation!

Schedules: “Good” vs “Bad”



Conflict serializability provides us with a notion of “good” vs “bad” schedules

Example: Not conflict serializable



T_1	T_2
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

- Equivalent to T_1, T_2, T_3, so serializable
- Not conflict equivalent to T_1, T_3, T_3 so not conflict serializable
- Conflict serializable => serializable but not the other way around!

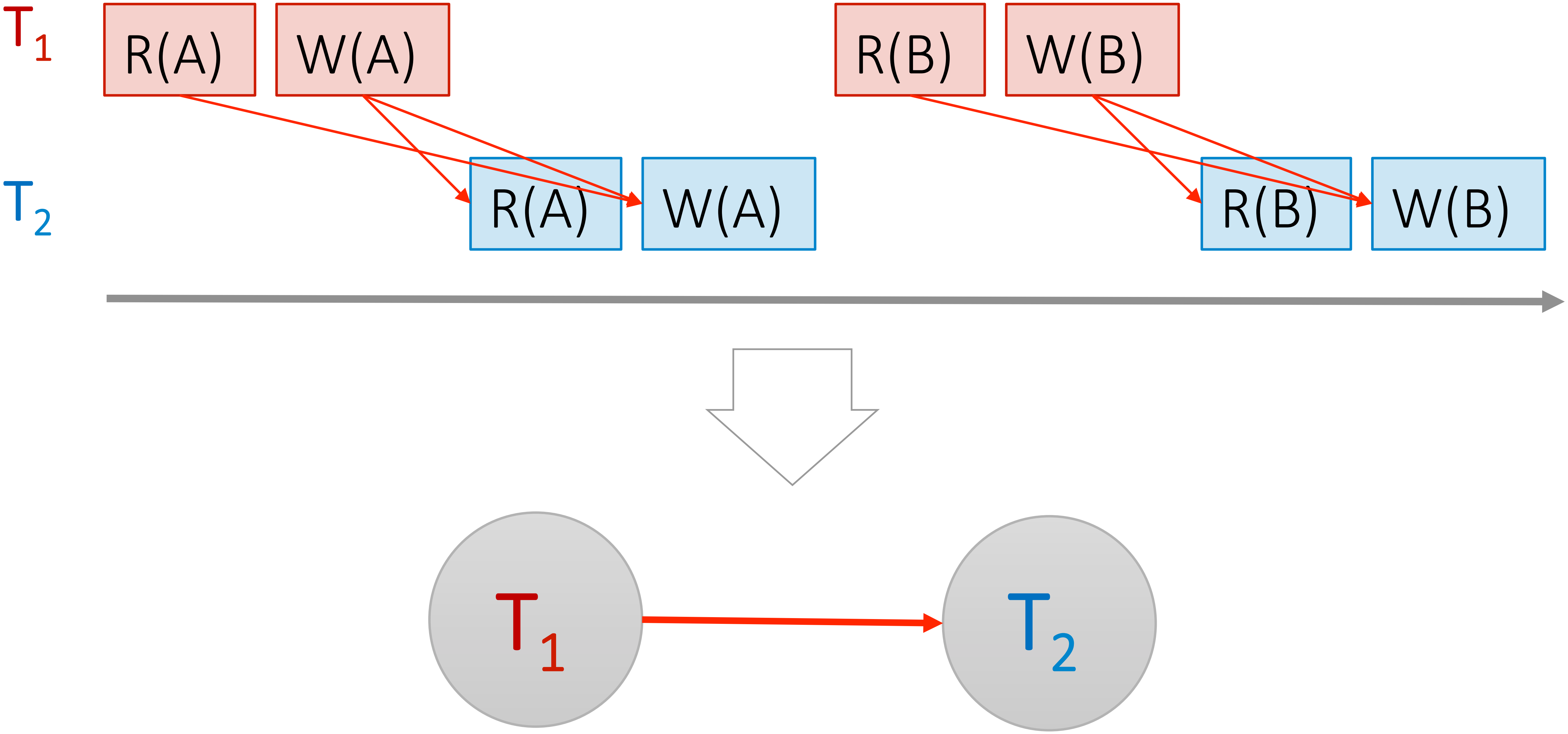
T_1	T_2	T_3
BEGIN Read(A);		
	BEGIN Write(A); COMMIT	
Write(A) COMMIT		
		BEGIN Write(A); COMMIT

Precedence (Serialization) Graph

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

Example: Precedence Graph

Serial Schedule:



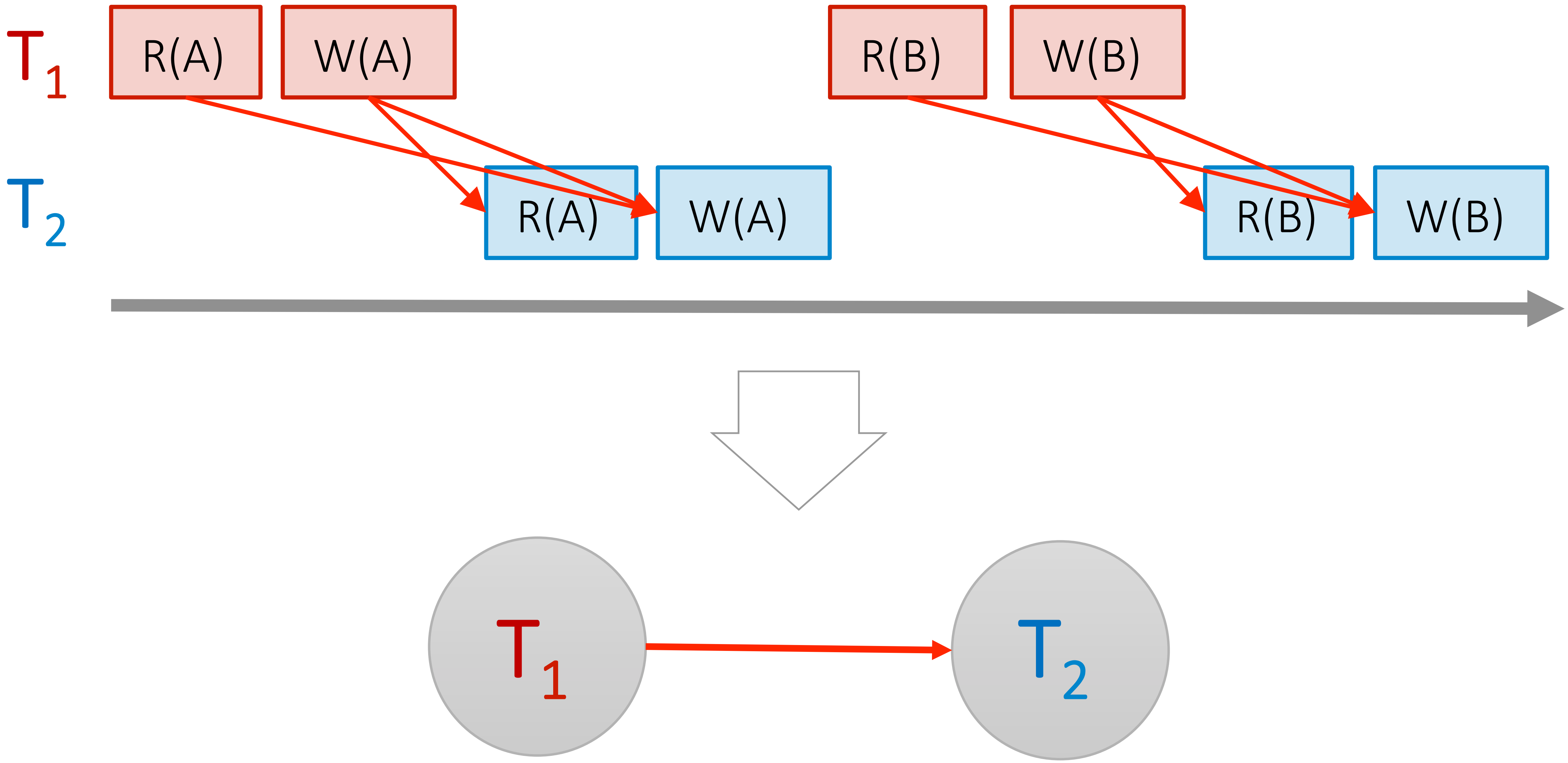
T_1: R(A) W(A) R(B) W(B)

T_2: R(A) W(A) R(B) W(B)

T_1 → T_2

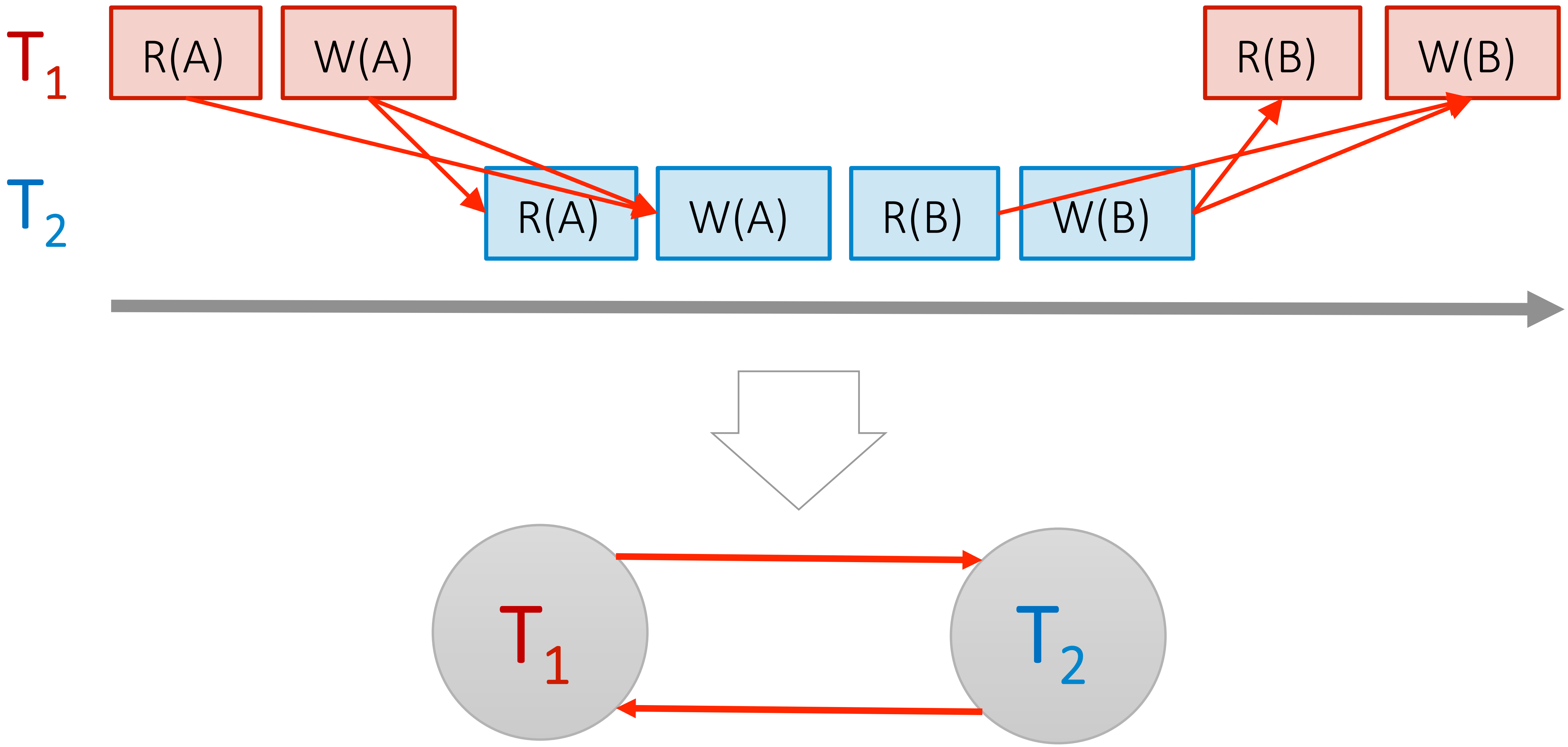
Example: Precedence Graph

Interleaved Schedule 1:



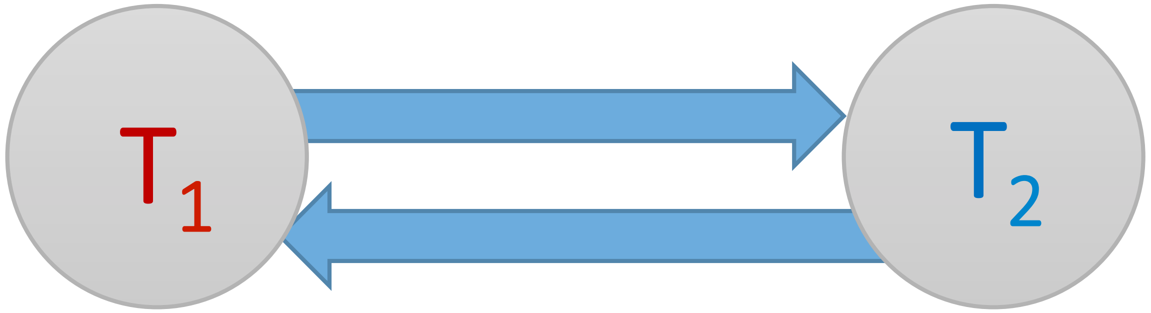
Example: Precedence Graph

Interleaved Schedule 2:



Example: Precedence Graph

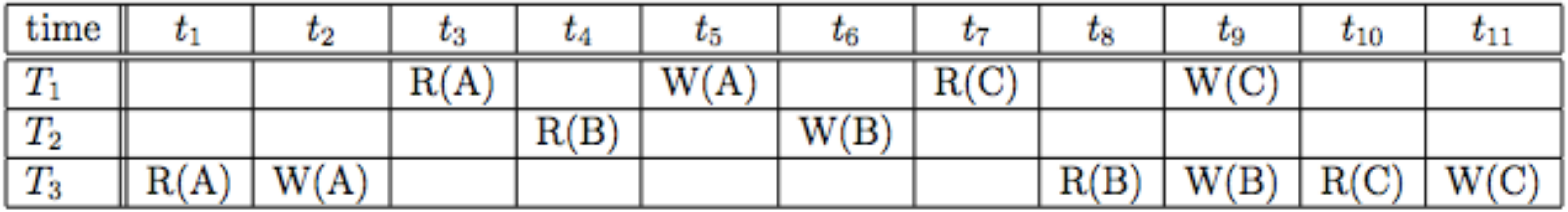
T_1	T_2
Read(A); A <— A + 100; Write(A);	
	Read(A); A <— A x 2; Write(A);
	Read(B); B <— B x 2; Write(B);
Read(B); B <— B + 100; Write(B);	



A non-conflict serializable schedule has a cycle!

Exercise: Serializability

- Consider the schedule given in the table below of three transactions T_1, T_2, and T_3



time	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
T_1			R(A)		W(A)		R(C)		W(C)		
T_2				R(B)		W(B)					
T_3	R(A)	W(A)						R(B)	W(B)	R(C)	W(C)

- Draw the precedence graph

- Is this schedule serializable?

Concurrency

- Schedules that are conflict serializable means that we are able to preserve isolation
- How can we guarantee conflict serializability in practice?
- What is the standard paradigm for concurrent programming?

Mutex <=> Lock <=> Semaphore

Locks: Basic Idea

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

T_1	T_2
BEGIN Lock(A) Read(A);	
	BEGIN Lock(A)
Write(A) Unlock(A) COMMIT	denied since T_1 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

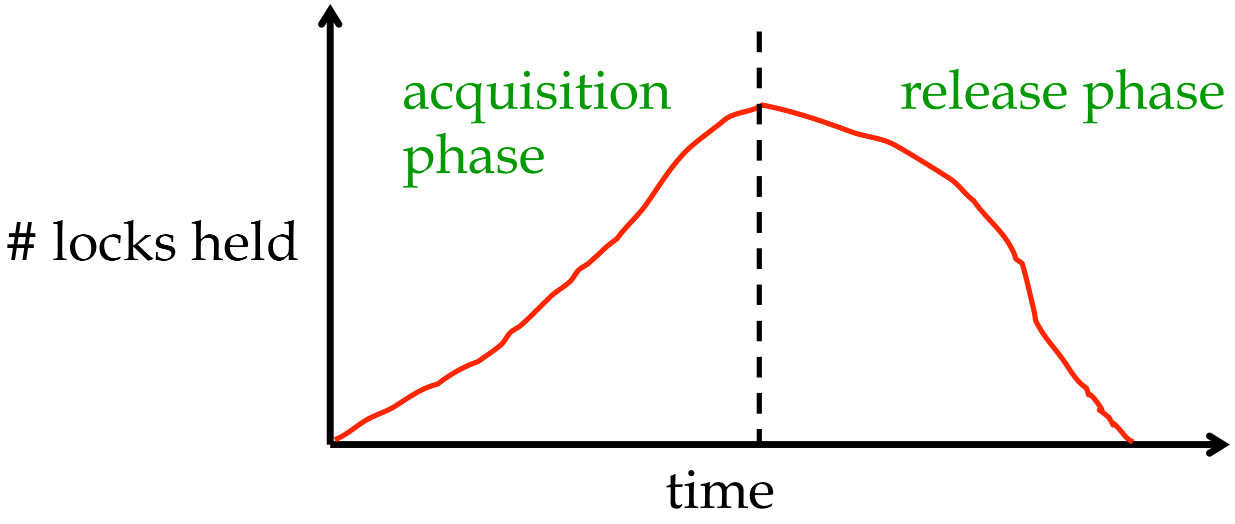
Does this work?

Example: Basic Locking Insufficient

	T_1	T_2	
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



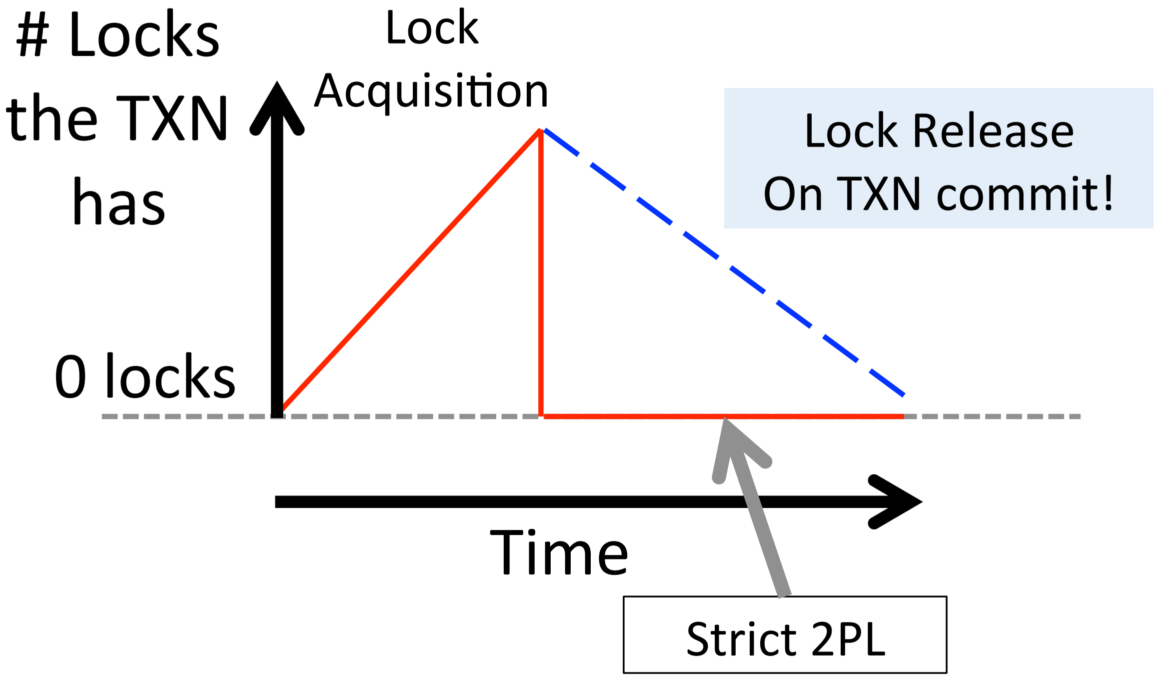
Does not prevent cascading aborts (where aborting one transaction causes one or more other transactions to abort)

Example: Cascading Abort

T_1	T_2
Exclusive-Lock(A); Read(A); A <— A + 5; Write(A); Exclusive-Lock(B) Unlock(A);	
cannot obtain lock on B until T_1 unlocks	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)	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



Strict 2PL: Properties

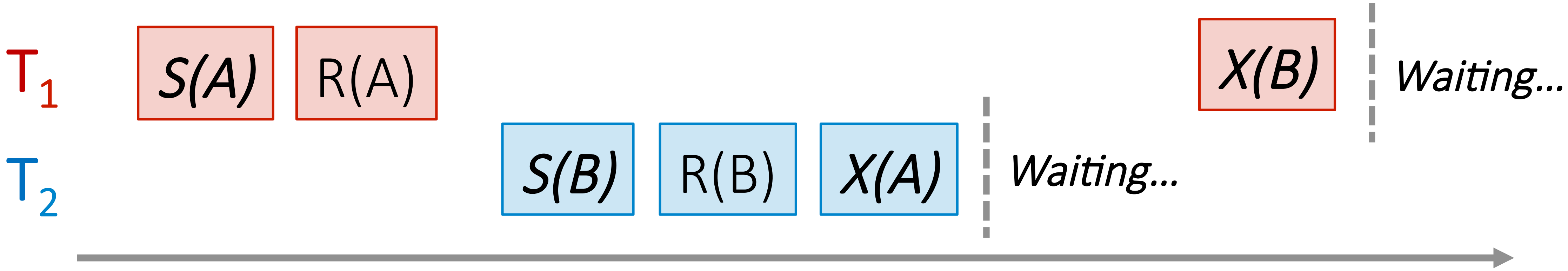
- Strict 2PL only allows conflict serializable schedules
    - Maintains serializable
    - Maintains isolation & consistency
- Used in many commercial DBMS systems
    - Oracle is notable exception

What could go wrong?

Example: Strict PL

T_1 requests shared lock on A to read it

T_1 requests exclusive lock on B to write



T_2 requests shared lock on B to read it

DEADLOCK!

T_2 requests exclusive lock on A to write

Deadlock

- Deadlock: Cycle of transactions waiting for locks to be released by each other
- Two ways of dealing with deadlocks
  - Deadlock prevention
  - Deadlock detection

Deadlock Protocols

- Deadlock prevention
  - Rigorous locking protocol — acquire all locks in advance
  - Timeout — waits some amount of time then roll back
- Deadlock detection
  - Construct waits-for graph (edge for any transaction waiting for another transaction) and periodically check for cycles

Transactions & Concurrency: Recap

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