Introduction to Database Systems
CSE 444

Lecture 14-15

Transactions: concurrency control (part 2)

Outline

- Continuing on locking (18.3)
- Isolation Levels
- Concurrency control by timestamps (18.8)
- Concurrency control by validation (18.9)

2PL Review

- 2PL enforces conflict-serializable schedules
- But what if a transaction releases its locks and then aborts?

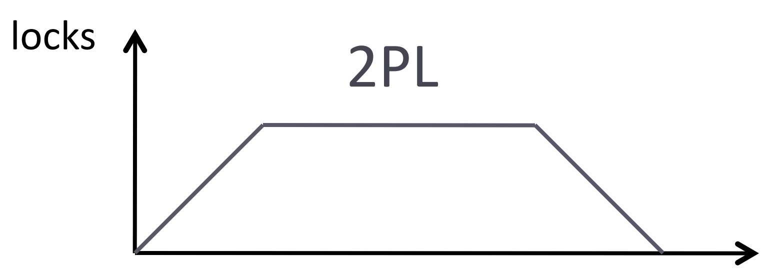
T1	T2
L_1(A); L_1(B); READ(A, t)	
t := t+100	
WRITE(A, t); U_1(A)	
	L_2(A); READ(A,s)
	s := s*2
	WRITE(A,s);
	L_2(B); **DENIED…**
READ(B, t)	
t := t+100	
WRITE(B,t); U_1(B);	
	…**GRANTED;** READ(B,s)
	s := s*2
	WRITE(B,s); U_2(A); U_2(B);
ABORT	

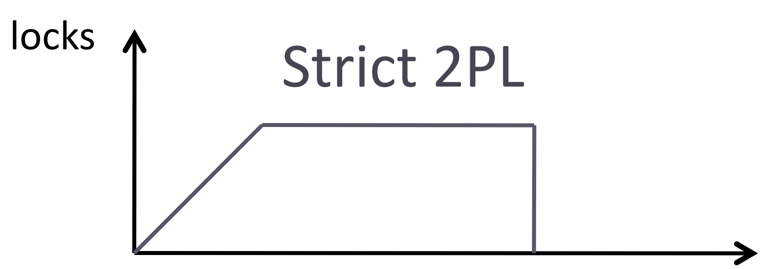
Now what? → ABORT

Strict 2PL

- Strict 2PL: All locks held by a transaction are released when the transaction is completed

- Ensures that schedules are recoverable
  - Transactions commit only after all transactions whose changes they read also commit
- Avoids cascading rollbacks





Deadlock

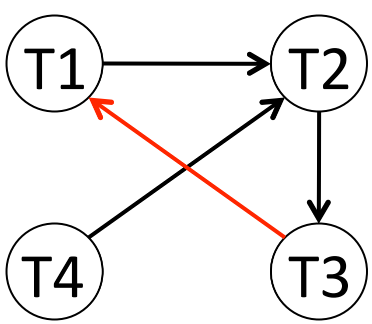
- Transaction T1 waits for a lock held by T2;
- But T2 waits for a lock held by T3;
- While T3 waits for . . . .
- . . .
- . . .and T73 waits for a lock held by T1 !!

Now what?

Deadlock: example

T1	T2	T3	T4
L(A)			
R(A)			
	L(B)		
	W(B)		
L(B)			
		L(C)	
		R(C)	
	L(C)		
			L(B)
		L(A)	

Waits-for graph



Deadlock!

Most systems do deadlock detection

Deadlock prevention

T_i requests a lock conflicting with T_j

- Wait-die:
  - If T_i has higher priority, it waits; otherwise it is aborted
- Wound-wait:
  - If T_i has higher priority, abort T_j; otherwise T_i waits

Conservative 2PL

- Acquire all locks at the beginning

Types of Locks

- Intuition: it's ok for many Xacts to read the same element.

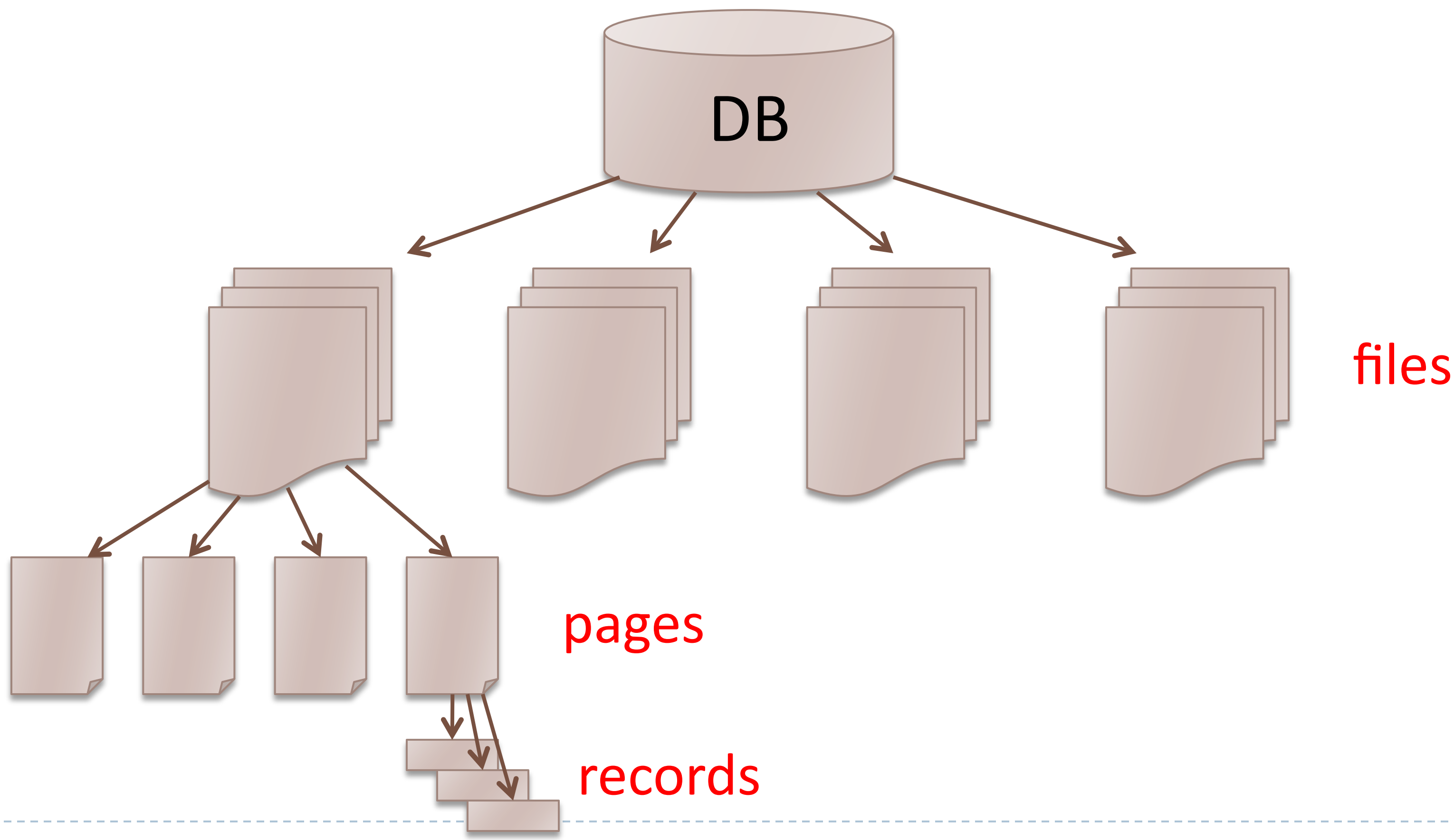
- Shared lock (S) – for reads
- Exclusive lock (X) – for writes

- Update lock (U) – initially S, possibly later upgrade to X

Mode	X	S	U
X	No	No	No
S	No	Yes	Yes
U	No	Yes	No

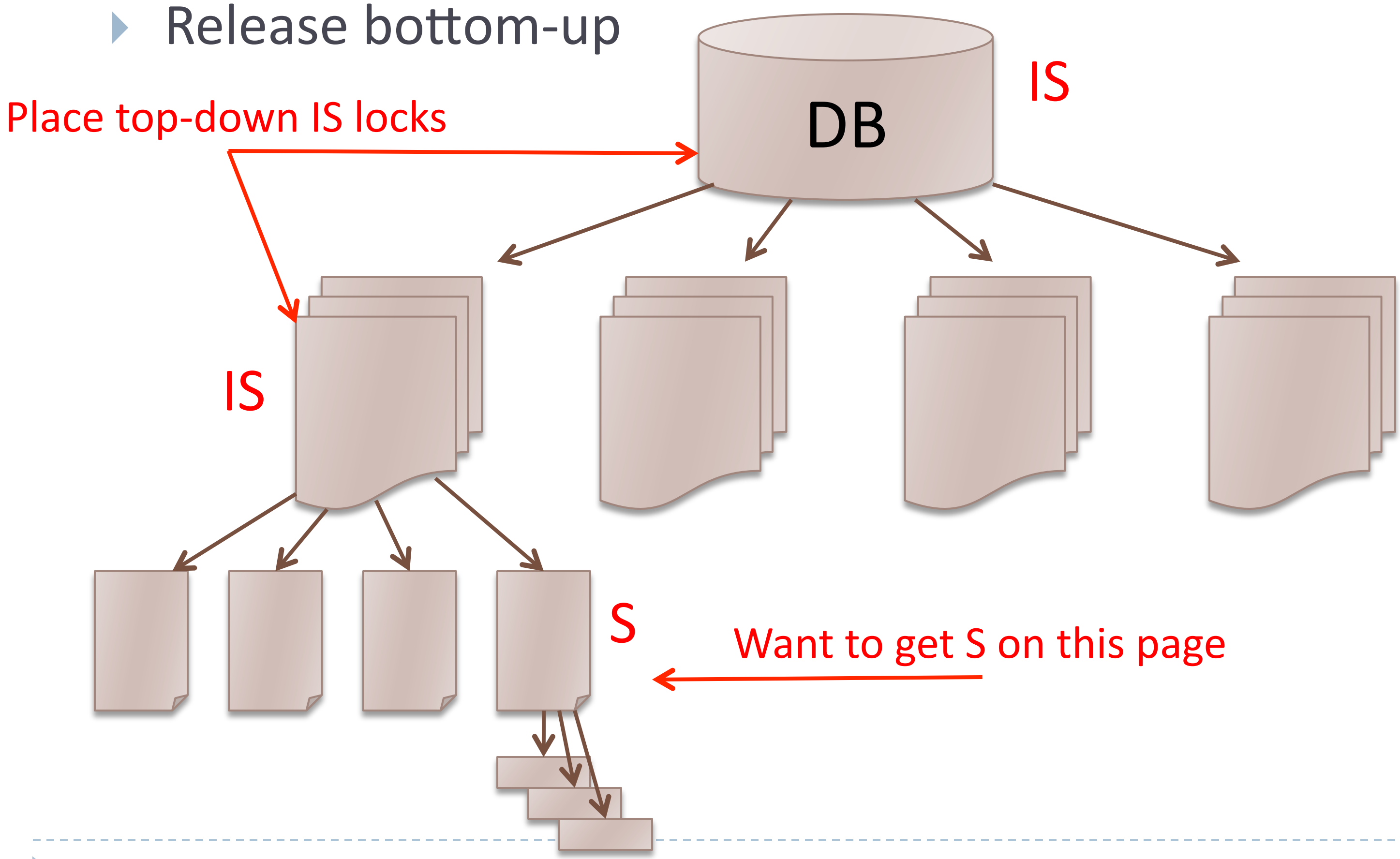
Granularity of Locks

- Multiple Granularity Locking
  - Allows locking of different size objects (files, pages, records)



Granularity of Locks

- Intention Locks: IS, IX, SIX
  - Lock with appropriate intention locks top down.
  - Release bottom-up



Granularity of Locks

Mode	IS	IX	S	SIX	U	X
IS	Yes	Yes	Yes	Yes	No	No
IX	Yes	Yes	No	No	No	No
S	Yes	No	Yes	No	Yes	No
SIX	Yes	No	No	No	No	No
U	No	No	Yes	No	No	No
X	No	No	No	No	No	No

Isolation Levels in SQL

- "Dirty reads"
  - SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
- "Committed reads"
  - SET TRANSACTION ISOLATION LEVEL READ COMMITTED
- "Repeatable reads"
  - SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
- Serializable transactions
  - SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

Choosing Isolation Level

- Trade-off: efficiency vs correctness

- DBMSs give user choice of level

Read DBMS docs!

Beware!!

- Default level is often NOT serializable
- Default level differs between DBMSs
- Some engines support subset of levels!

1. Isolation Level: Dirty Reads

Implementation using locks:

- "Long duration" WRITE locks
  - A.k.a Strict Two Phase Locking (you knew that !)
- Do not use READ locks
  - Read-only transactions are never delayed

- Possible problems: dirty and inconsistent reads

2. Isolation Level: Read Committed

Implementation using locks:

- "Long duration" WRITE locks
- "Short duration" READ locks
  - Only acquire lock while reading (not 2PL)

- Possible problems: unrepeatable reads
  - When reading same element twice,
  - may get two different values

3. Isolation Level: Repeatable Read

Implementation using locks:

- "Long duration" READ and WRITE locks
  - Full Strict Two Phase Locking

- This is not serializable yet !!!

What could be the problem??

The Phantom Problem

- We've been looking at updates
  - What about insertions/deletions?

```
T1:
   select count(*) from R where price>20
   . . . .
   . . . .
   . . . .
   . . . .
   select count(*) from R where price>20
```

```
T2:
   . . . .
   . . . .
   insert into R(name,price)
         values('Gizmo', 50)
   . . . .
```

Aha! Phantom tuple!

Solutions:
- Coarse locks (table level)
- Predicate locking (index locking)

Isolation levels: Summary

Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom Read**
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Beyond Locking

- Optimistic Concurrency Control

- Intuition:
  - There is overhead in locking, so if we don't expect may conflicts, we can sort of "wing it" and hope for the best ☺

Timestamps

- Each transaction receives a unique timestamp TS(T)

- Could be:
  - The system's clock
  - A unique counter, incremented by the scheduler

Timestamps

Main invariant:

The timestamp order defines the serialization order of the transaction

Main Idea

- For any two conflicting actions, ensure that their order is the serialized order:
- In each of these cases
  - $W_{T1}(X) \ldots R_{T2}(X)$
  - $R_{T1}(X) \ldots W_{T2}(X)$
  - $W_{T1}(X) \ldots W_{T2}(X)$

  } Possible conflicts

- Answer: Check that TS(T1) < TS(T2)

When T2 wants to read X, $r_{T2}(X)$, how do we know T1, and TS(T1) ?

Timestamps

With each element X, associate:

- RT(X) = the highest timestamp of any transaction that read X
- WT(X) = the highest timestamp of any transaction that wrote X
- C(X) = the commit bit: true when transaction with highest timestamp that wrote X committed

If 1 element = 1 page, these are associated with each page X in the buffer pool

Time-based Scheduling

Note: simple version that ignores the commit bit

- Transaction wants to read element X
  - If TS(T) < WT(X) abort
  - Else read and update RT(X) to larger of TS(T) or RT(X)

- Transaction wants to write element X
  - If TS(T) < RT(X) abort
  - Else if TS(T) < WT(X) ignore write & continue (Thomas Write Rule)
  - Otherwise, write X and update WT(X) to TS(T)

Details

Read too late:

- T1 wants to read X, and TS(T1) < WT(X)

START(T1) … START(T2) … $W_{T2}(X)$ . . . $R_{T1}(X)$

Need to rollback T1!

Details

Write too late:

- T1 wants to write X, and $TS(T1) < RT(X)$

$$START(T1) \ldots START(T2) \ldots R_{T2}(X) \ . \ . \ . \ W_{T1}(X)$$

Need to rollback T1!

Details

Write too late, but we can still handle it:

- T1 wants to write X, and
  TS(T1) ≥ RT(X) but WT(X) > TS(T1)

START(T1) … START(T2) … $W_{T2}(X)$ . . . $W_{T1}(X)$

Don’t write X at all!

More Problems

Read dirty data:

- T2 wants to read X, and WT(X) < TS(T2)
- Seems OK, but…

START(T1) … START(T2) … W_{T1}(X). . . R_{T2}(X)… ABORT(T1)

If C(X)=false, T2 needs to wait for it to become true

More Problems

Write dirty data:

- T1 wants to write X, and WT(X) > TS(T1)
- Seems OK not to write at all, but …

START(T1) … START(T2) … W_{T2}(X). . . W_{T1}(X)… ABORT(T2)

If C(X)=false, T1 needs to wait for it to become true

Timestamp-based Scheduling

- When a transaction T requests R(X) or W(X), the scheduler examines RT(X), WT(X), C(X), and decides one of:
  - To grant the request, or
  - To rollback T (and restart) ← With what timestamp?
  - To delay T until C(X) = true

Timestamp-based Scheduling

RULES including commit bit

- There are 4 long rules in Sec. 18.8.4
- You should be able to derive them yourself, based on the previous slides

READING ASSIGNMENT: 18.8.4

Multiversion Timestamp

- When transaction T requests R(X) but WT(X) > TS(T), then T must rollback

- Idea: keep multiple versions of X:
  $X_t, X_{t-1}, X_{t-2}, \ldots$

$$TS(X_t) > TS(X_{t-1}) > TS(X_{t-2}) > \ldots$$

- Let T read an older version, with appropriate timestamp

Details

- When $W_T(X)$ occurs,
  - create a new version, denoted X_t where t = TS(T)
- When $R_T(X)$ occurs,
  - find most recent version X_t such that t < TS(T)
  - Notes:
    - $WT(X_t)$ = t and it never changes
    - $RT(X_t)$ must still be maintained to check legality of writes
- Can delete X_t if we have a later version X_{t1} and all active transactions T have TS(T) > t1

Tradeoffs

- Locks:
  - Great when there are many conflicts
  - Poor when there are few conflicts
- Timestamps
  - Poor when there are many conflicts (rollbacks)
  - Great when there are few conflicts

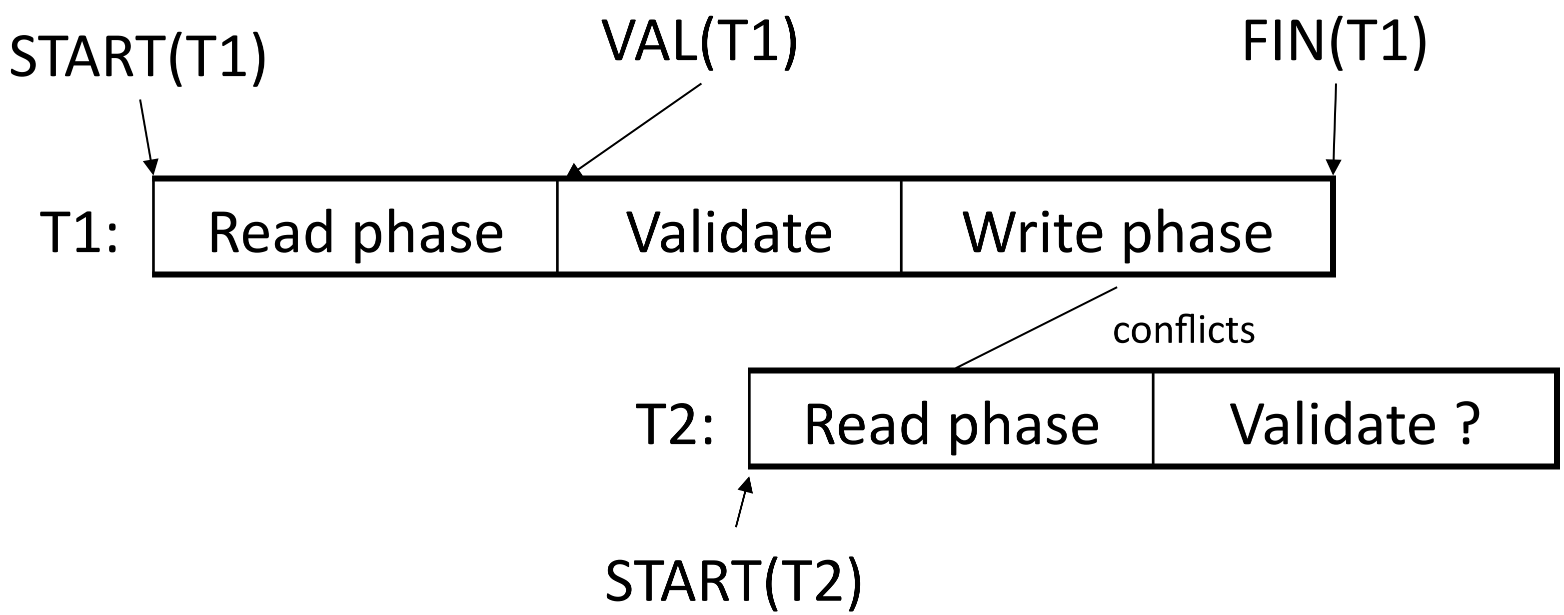
- Compromise
  - READ ONLY transactions → timestamps
  - READ/WRITE transactions → locks

Concurrency Control by Validation

- Each transaction T defines a read set RS(T) and a write set WS(T)
- Each transaction proceeds in three phases:
  - Read all elements in RS(T). Time = START(T)
  - Validate (may need to rollback). Time = VAL(T)
  - Write all elements in WS(T). Time = FIN(T)

Main invariant: the serialization order is VAL(T)

Avoid $R_{T2}(X) - W_{T1}(X)$ Conflicts

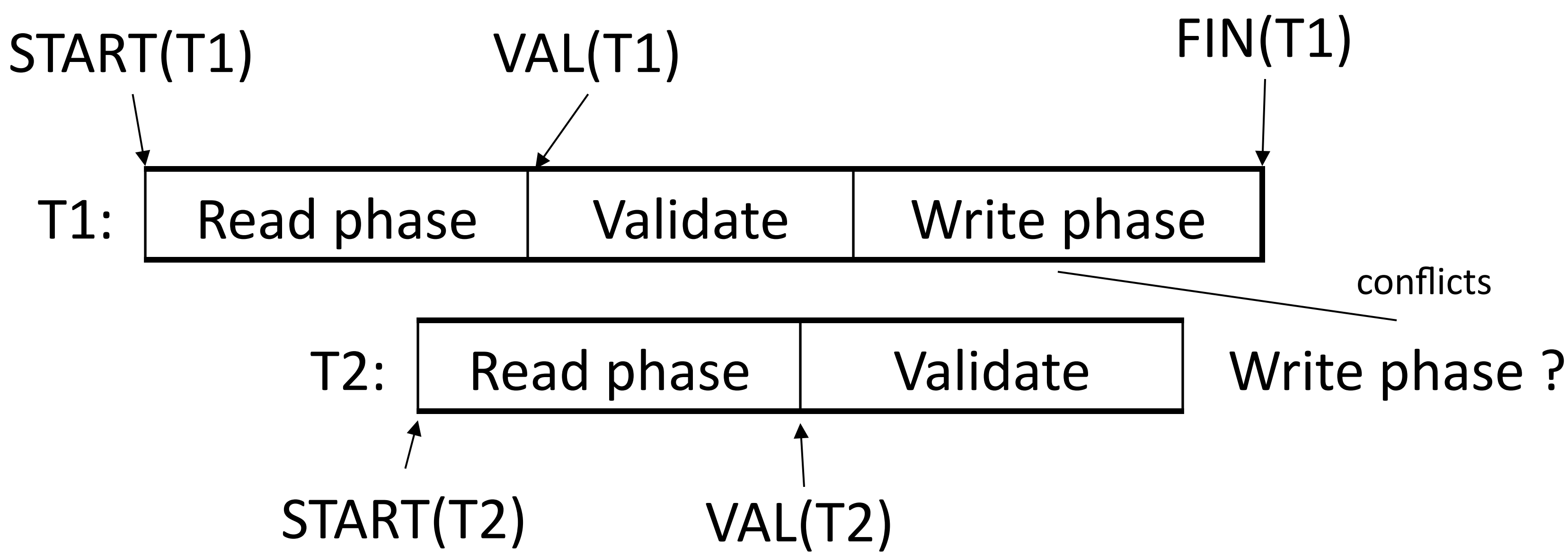


If RS(T2)∩WS(T1) not empty and FIN(T1) > START(T2)

(T1 has validated and T1 has not finished before T2 begun)

Then ROLLBACK(T2)

Avoid $W_{T2}(X) - W_{T1}(X)$ Conflicts



If WS(T2)∩WS(T1) not empty and FIN(T1) > VAL(T2)

(T1 has validated and T1 has not finished before T2 validates)

Then ROLLBACK(T2)