Database System Internals
Optimistic Concurrency Control

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

About Lab 3

- **In lab 3, we implement transactions**
- Focus on concurrency control
  - Want to run many transactions at the same time
  - Transactions want to read and write same pages
  - Will use locks to ensure conflict serializable execution
  - Use strict 2PL

- Build your own lock manager
  - Understand how locking works in depth
  - Ensure transactions rather than threads hold locks
    - Many threads can execute different pieces of the same transaction
    - Need to detect deadlocks and resolve them by aborting a transaction
  - But use Java synchronization to protect your data structures

Recap

- Several types of schedules:
  - Serializable, conflict serializable, view serializable
  - Recoverable, without cascading aborts

- 2PL guarantees conflict serializable schedules

- Strict 2PL also guarantees no-cascading-aborts

- Locking manager: inserts lock/unlock, manages locks

- Types of locks: shared, exclusive

Isolation Levels in SQL

1. "Dirty reads"
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. "Committed reads"
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. "Repeatable reads"
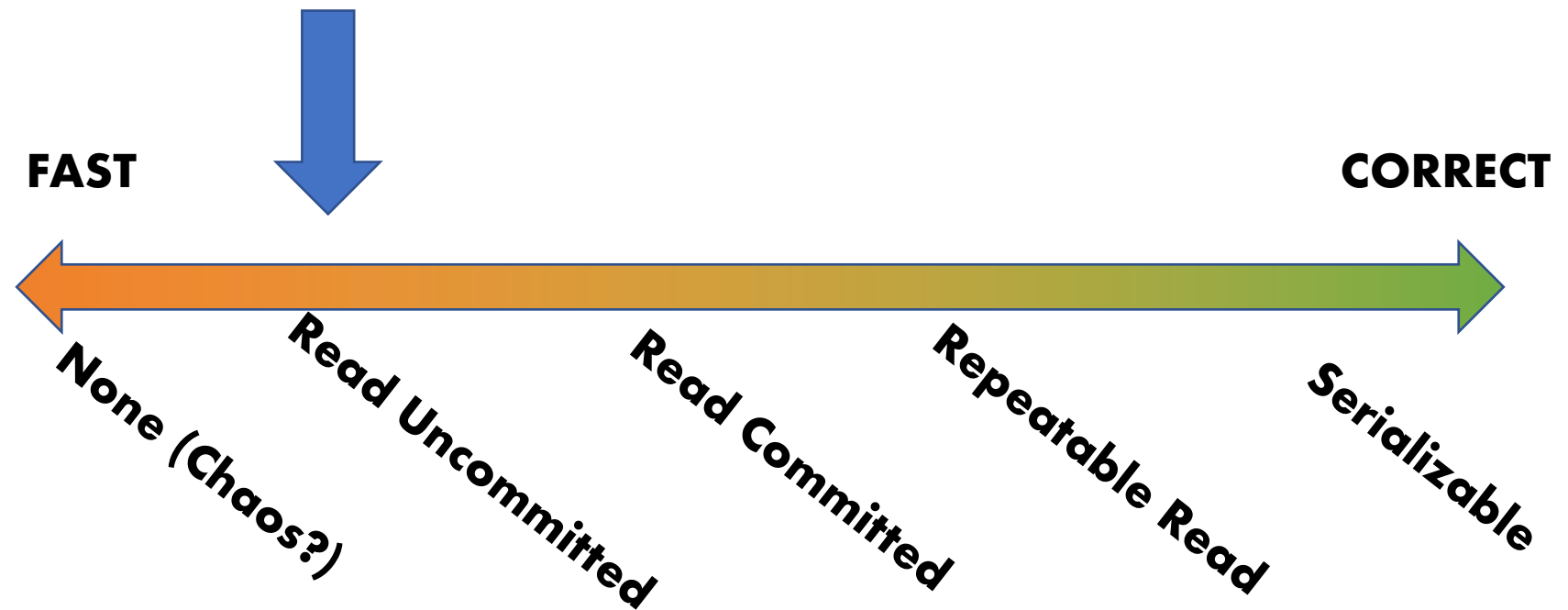   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

ACID

Isolation Level Design Spectrum

FAST   **CORRECT**

None (Chaos?)   Read Uncommitted   Read Committed   Repeatable Read   Serializable

1. Isolation Level: Dirty Reads

- **"Long duration" WRITE locks**
  - Strict 2PL

- **No READ locks**
  - Read-only transactions are never delayed

Possible problems:
  dirty and inconsistent reads

2. Isolation Level: Read Committed

- **"Long duration" WRITE locks**
  - Strict 2PL

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

Unrepeatable reads
    When reading same element twice,
    may get two different values

3. Isolation Level: Repeatable Read

- **"Long duration" WRITE locks**
  - Strict 2PL

- **"Long duration" READ locks**
  - Strict 2PL

This is not serializable yet !!!

Why ?

4. Isolation Level Serializable

- **"Long duration" WRITE locks**
  - Strict 2PL
- **"Long duration" READ locks**
  - Strict 2PL
- **Predicate locking**
  - To deal with phantoms

READ-ONLY Transactions

Client 1: START TRANSACTION
    INSERT INTO SmallProduct(name, price)
        SELECT pname, price
        FROM Product
        WHERE price <= 0.99

    DELETE FROM Product
        WHERE price <=0.99
    COMMIT

Client 2: SET TRANSACTION READ ONLY
    START TRANSACTION
    SELECT count(*)
    FROM Product

    SELECT count(*)
    FROM SmallProduct
    COMMIT

May improve performance

Commercial Systems

Always check documentation!

- **DB2**: Strict 2PL

- **SQL Server**:
  - Strict 2PL for standard 4 levels of isolation
  - Multiversion concurrency control for snapshot isolation

- **PostgreSQL:** Snapshot isolation; recently: seralizable Snapshot isolation (!)

- **Oracle:** Snapshot isolation

Pessimistic vs. Optimistic
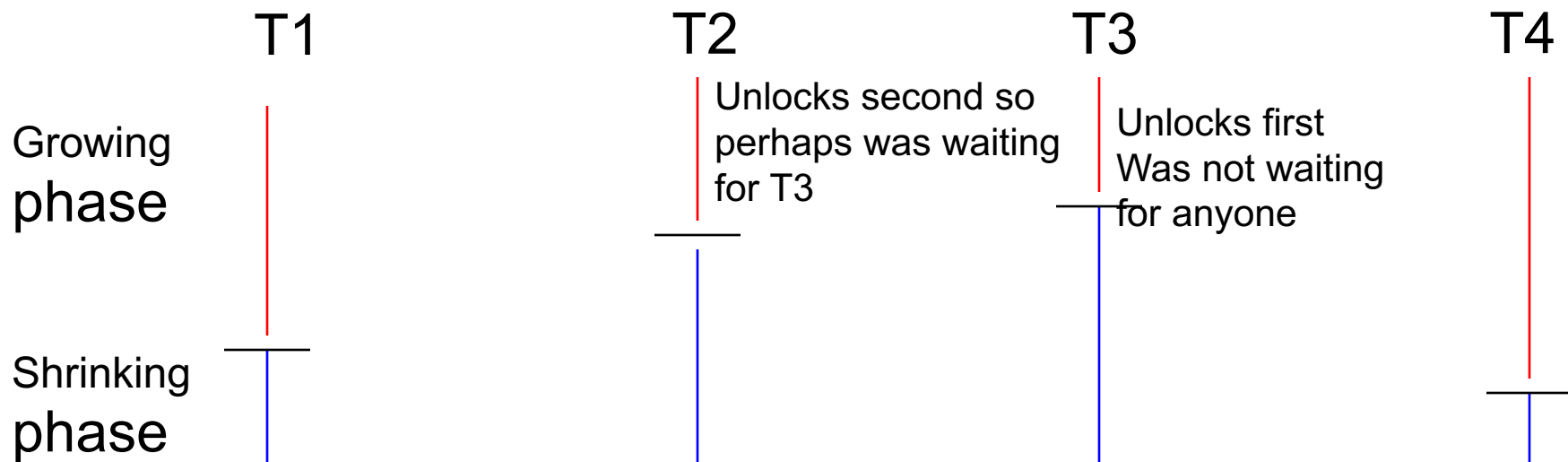
- ## Pessimistic CC (locking)
    - Prevents unserializable schedules
    - Never abort for serializability (but may abort for deadlocks)
    - Best for workloads with high levels of contention

- ## Optimistic CC (timestamp, multi-version, validation)
    - Assume schedule will be serializable
    - Abort when conflicts detected
    - Best for workloads with low levels of contention

Outline

- **Concurrency control by timestamps (18.8)**

- Concurrency control by validation (18.9)

- Snapshot Isolation

Example with Multiple Transactions

T1

T2   Unlocks second so
      perhaps was waiting
      for T3

T3   Unlocks first
      Was not waiting
      for anyone

T4

Growing
phase

Shrinking
phase

Equivalent to each transaction executing entirely the moment it enters shrinking phase

Timestamps

- Each transaction receives unique timestamp TS(T)

Could be:

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

Timestamps

Main invariant:

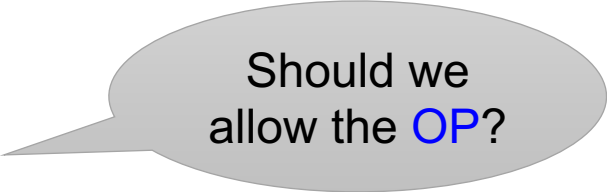The timestamp order defines
the serialization order of the transaction

Will generate a schedule that is view-equivalent
to a serial schedule, and recoverable

Main Idea

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$

- Should it allow it to proceed? Wait? Abort?

- Consider these cases:

$$w_U(X) \ldots r_T(X)$$
$$r_U(X) \ldots w_T(X)$$
$$w_U(X) \ldots w_T(X)$$

Should we allow the OP?

Main Idea

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

$$w_U(X) \ldots \ r_T(X)$$
$$r_U(X) \ldots \ w_T(X)$$
$$w_U(X) \ldots \ w_T(X)$$

Should we allow the OP?

$$\text{START(U)}, \ldots, \text{START(T)}, \ldots, w_U(X), \ldots, r_T(X)$$

Main Idea

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

$$w_U(X) \ldots r_T(X)$$
$$r_U(X) \ldots w_T(X)$$
$$w_U(X) \ldots w_T(X)$$

Should we allow the OP?

OK

$$\text{START(U), ...,START(T), ..., } w_U(X), ..., r_T(X)$$

Main Idea

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$

- Should it allow it to proceed? Wait? Abort?

- Consider these cases:

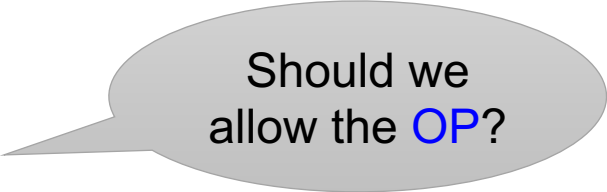$$w_U(X) \ldots \ r_T(X)$$
$$r_U(X) \ldots \ w_T(X)$$
$$w_U(X) \ldots \ w_T(X)$$

Should we allow the OP?

OK

$$\text{START}(U), ..., \text{START}(T), ..., w_U(X), ..., r_T(X)$$

$$\text{START}(T), ..., \text{START}(U), ..., w_U(X), ..., r_T(X)$$

Main Idea

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$

- Should it allow it to proceed? Wait? Abort?

- Consider these cases:

$$w_U(X) \ldots \quad r_T(X)$$
$$r_U(X) \quad \ldots \quad w_T(X)$$
$$w_U(X) \ldots \quad w_T(X)$$

Should we allow the OP?

$$\text{START(U), ...,START(T), ..., } w_U(X), ..., r_T(X)$$

OK

$$\text{START(T), ...,START(U), ..., } w_U(X), ..., r_T(X)$$

Too late

Timestamps

With each element X, associate

- RT(X) = the highest timestamp of any transaction U that read X

- WT(X) = the highest timestamp of any transaction U that wrote X

- C(X) = the commit bit: true when transaction with highest timestamp that **wrote** X committed

Timestamps

With each element X, associate
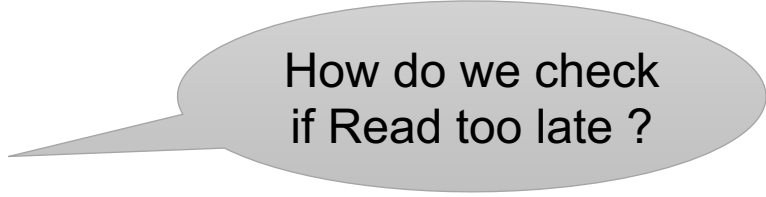
- RT(X) = the highest timestamp of any transaction U that read X

- WT(X) = the highest timestamp of any transaction U that wrote X

- C(X) = the commit bit: true when transaction with highest timestamp that **wrote** X committed

If transactions abort, we must **reset the timestamps**
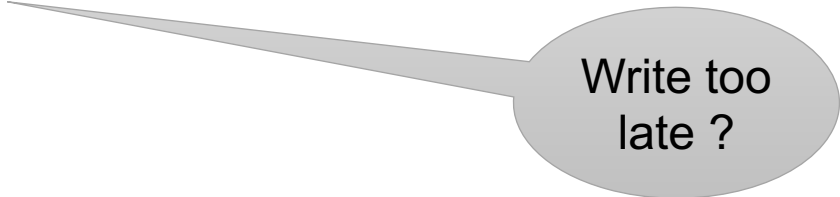
Main Idea

For any $r_T(X)$ or $w_T(X)$ request, check for conflicts:

- $w_U(X) \ldots r_T(X)$
- $r_U(X) \ldots w_T(X)$
- $w_U(X) \ldots w_T(X)$

How do we check
if Read too late ?
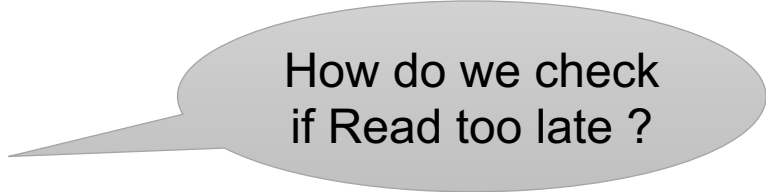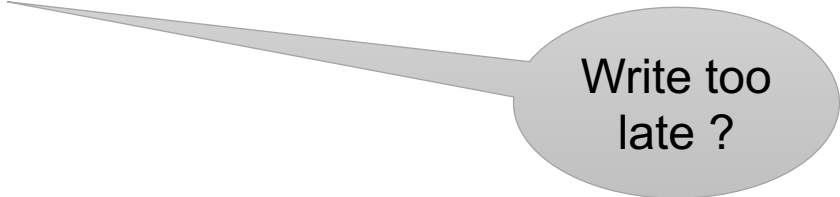
Write too
late ?

Main Idea

For any $r_T(X)$ or $w_T(X)$ request, check for conflicts:

- $w_U(X) \ldots r_T(X)$
- $r_U(X) \ldots w_T(X)$
- $w_U(X) \ldots w_T(X)$

How do we check
if Read too late ?

Write too
late ?

When T requests $r_T(X)$, need to check $TS(U) \le TS(T)$

Read Too Late?

- T wants to read X

$$\text{START(T)} \dots \text{START(U)} \dots w_U(X) \dots r_T(X)$$

Read Too Late?
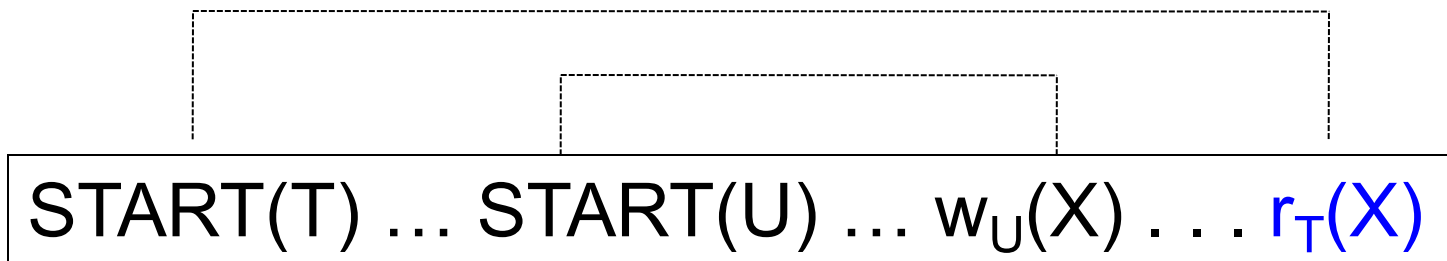
- T wants to read X

$$\text{START(T) ... START(U) ... } w_U(X) \ldots r_T(X)$$
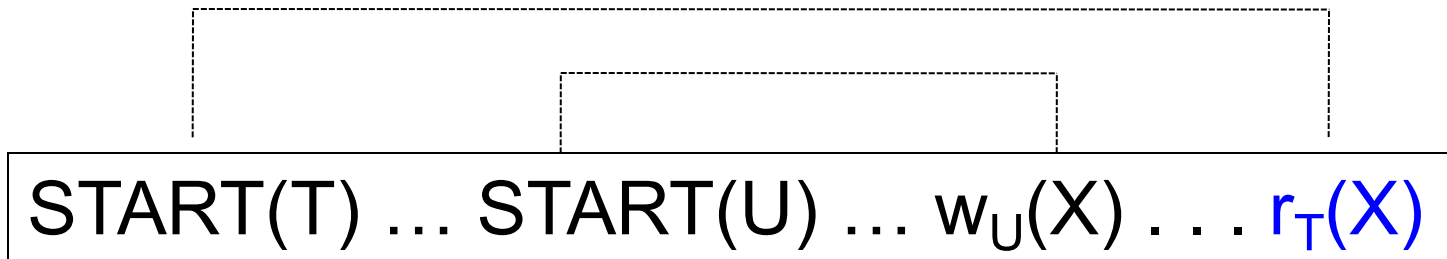
If WT(X) > TS(T) then need to rollback T !
T tried to read **too late**

Simplified TS-based Schedule (no Aborts)

Request is $r_T(X)$
    ??

Simplified TS-based Schedule (no Aborts)

Request is $r_T(X)$
> If $WT(X) > TS(T)$ then ROLLBACK
> Else READ and update $RT(X)$ to larger of $TS(T)$ or $RT(X)$
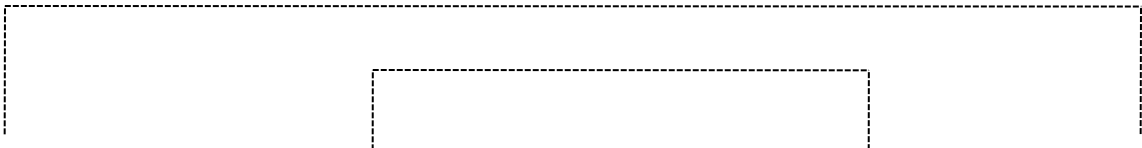
Write Too Late?

- T wants to write X

$$\text{START(T)} \ldots \text{START(U)} \ldots r_U(X) \ldots w_T(X)$$

Write Too Late?

- T wants to write X

$$\text{START(T)} \ldots \text{START(U)} \ldots r_U(X) \ldots w_T(X)$$

If $RT(X) > TS(T)$ then need to rollback T !
T tried to write **too late**

Simplified TS-based Schedule (no Aborts)

Request is $r_T(X)$
    If $WT(X) > TS(T)$ then ROLLBACK
    Else READ and update RT(X) to larger of TS(T) or RT(X)

Request is $w_T(X)$
    ???

Simplified TS-based Schedule (no Aborts)

Request is $r_T(X)$

    If $WT(X) > TS(T)$ then ROLLBACK

    Else READ and update RT(X) to larger of TS(T) or RT(X)

Request is $w_T(X)$

    If $RT(X) > TS(T)$ then ROLLBACK

    what about WT(X)?

    Otherwise, WRITE and update WT(X) =TS(T)

Thomas' Rule

But… we can still handle it in one case:

- T wants to write X

$$\text{START}(T) \ldots \text{START}(V) \ldots w_V(X) \ldots w_T(X)$$

Thomas' Rule

But we can still handle it:

- T wants to write X

Is this conflict-serializable?

$$\text{START(T) … START(V) … } w_V(X) \text{ . . . } w_T(X)$$

If $RT(X) \le TS(T)$ and $WT(X) > TS(T)$
then don't write X at all !

Thomas' Rule

But we can still handle it:

- T wants to write X

Is this conflict-serializable?

$$\text{START(T)} \ldots \text{START(V)} \ldots w_V(X) \ldots w_T(X)$$

If $RT(X) \leq TS(T)$ and $WT(X) > TS(T)$
then don't write X at all !

View serializable!

Simplified TS-based Schedule (no Aborts)

Request is $r_T(X)$

    If $WT(X) > TS(T)$ then ROLLBACK
    Else READ and update RT(X) to larger of TS(T) or RT(X)

Request is $w_T(X)$

    If $RT(X) > TS(T)$ then ROLLBACK
    what about WT(X)?

    Otherwise, WRITE and update WT(X) =TS(T)

Simplified TS-based Schedule (no Aborts)

Request is $r_T(X)$
    If $WT(X) > TS(T)$ then ROLLBACK
    Else READ and update RT(X) to larger of TS(T) or RT(X)

Request is $w_T(X)$
    If $RT(X) > TS(T)$ then ROLLBACK
    Else if $WT(X) > TS(T)$ ignore write & continue
        (Thomas Write Rule)
    Otherwise, WRITE and update WT(X) =TS(T)

View-serializable

Simplified TS-based Schedule (no Aborts)

- The simplified timestamp-based scheduling with Thomas' rule ensures that the schedule is view-serializable

Ensuring Recoverable Schedules

Recall:

- Schedule without cascading aborts: when a transaction reads an element, then transaction that wrote it must have already committed

- Use the commit bit C(X) to keep track if the transaction that **last wrote** X has committed
  (just a read will not change the commit bit)

Ensuring Recoverable Schedules

Read dirty data:

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

START(U) … START(T) … $w_U(X)$. . . . $r_T(X)$ … ABORT(U)
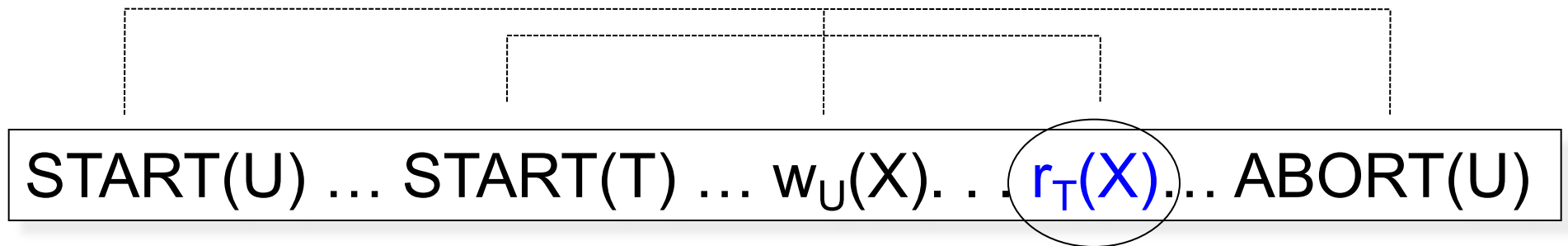
If C(X)=false, T needs to wait for it to become true

Ensuring Recoverable Schedules

Thomas' rule needs to be revised:

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

START(T) … START(U)… $w_U(X)$. . . $w_T(X)$… ABORT(U)
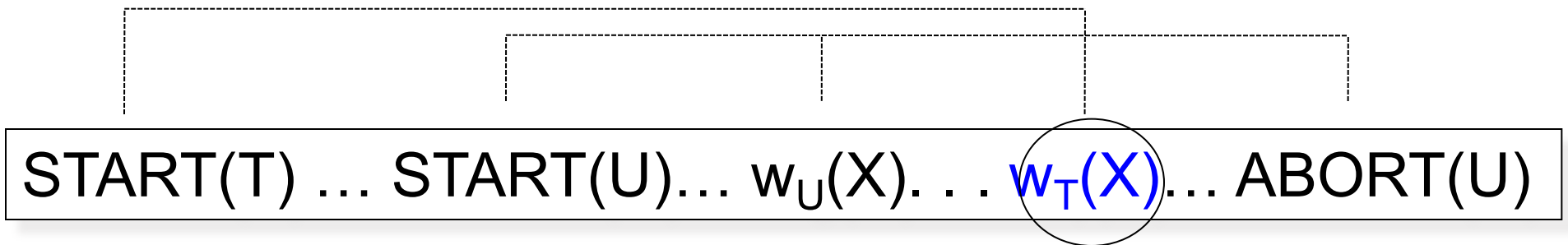
If C(X)=false, T needs to wait for it to become true

Timestamp-based Scheduling

- When a transaction T requests $r_T(X)$ or $w_T(X)$, the scheduler examines RT(X), WT(X), C(X), and decides one of:

- To grant the request, or
- To rollback T (and restart with later 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
- Make sure you understand them !

READING ASSIGNMENT:

Garcia-Molina et al. 18.8.4

Timestamp-based Scheduling (sec. 18.8.4)

Transaction wants to READ element X
    If $WT(X) > TS(T)$ then ROLLBACK
    Else If $C(X)$ = false, then WAIT
    Else READ and update $RT(X)$ to larger of $TS(T)$ or $RT(X)$

Transaction wants to WRITE element X
    If $RT(X) > TS(T)$ then ROLLBACK
    Else if $WT(X) > TS(T)$
        Then If $C(X)$ = false then WAIT
            else IGNORE write (Thomas Write Rule)
    Otherwise, WRITE, and update $WT(X)=TS(T)$, $C(X)$=false

Basic Timestamps with Commit Bit

T₁	T₂	T₃	T₄	A
1	2	3	4	RT=0
				WT=0 C=true

Time

Basic Timestamps with Commit Bit

T$_1$	T$_2$	T$_3$	T$_4$	A
1	2	3	4	RT=0
				WT=0 C=true
	W$_2$(A)			

Time

Basic Timestamps with Commit Bit

T$_1$	T$_2$	T$_3$	T$_4$	A
1	2	3	4	RT=0
				WT=0 C=true
	W$_2$(A)			WT=2 C=false
				RT=0

Time

Basic Timestamps with Commit Bit

T$_1$	T$_2$	T$_3$	T$_4$	A
1	2	3	4	RT=0
				WT=0 C=true
	W$_2$(A)			WT=2 C=false
R$_1$(A)				RT=0

Time

Basic Timestamps with Commit Bit

T$_1$	T$_2$	T$_3$	T$_4$	A
1	2	3	4	RT=0
				WT=0 C=true
	W$_2$(A)			WT=2 C=false
				RT=0
R$_1$(A)				
Abort				

Time ↓

Basic Timestamps with Commit Bit

T$_1$	T$_2$	T$_3$	T$_4$	A
1	2	3	4	RT=0
				WT=0 C=true
	W$_2$(A)			WT=2 C=false
R$_1$(A)				RT=0
Abort		R$_3$(A)		

Time

Basic Timestamps with Commit Bit

	T$_1$	T$_2$	T$_3$	T$_4$	A	
	1	2	3	4	RT=0	
					WT=0	C=true
		W$_2$(A)			WT=2	C=false
Time	R$_1$(A) **Abort**				RT=0	
			R$_3$(A) **Delay**			

Basic Timestamps with Commit Bit

T$_1$	T$_2$	T$_3$	T$_4$	A
1	2	3	4	RT=0
				WT=0 C=true
	W$_2$(A)			WT=2 C=false
				RT=0
R$_1$(A)				
Abort		R$_3$(A)		
		Delay		
	C			

Time

Basic Timestamps with Commit Bit

	T$_1$	T$_2$	T$_3$	T$_4$	A
	1	2	3	4	RT=0
					WT=0 C=true
		W$_2$(A)			WT=2 C=false
					RT=0
Time	R$_1$(A) **Abort**		R$_3$(A) **Delay**		
		C			C=true

Basic Timestamps with Commit Bit

T₁	T₂	T₃	T₄	A
T_1	T_2	T_3	T_4	A
1	2	3	4	RT=0
				WT=0 C=true
	$W_2(A)$			WT=2 C=false
				RT=0
$R_1(A)$				
Abort		$R_3(A)$		
		Delay		
	C			C=true
		$R_3(A)$		

Time

Basic Timestamps with Commit Bit

T₁	T₂	T₃	T₄	A
1	2	3	4	RT=0
				WT=0 C=true
	W₂(A)			WT=2 C=false
				RT=0
R₁(A)				
Abort		R₃(A)		
		Delay		
	C			C=true
		R₃(A)		RT=3

Time ↓

Basic Timestamps with Commit Bit

	T_1	T_2	T_3	T_4	A
	1	2	3	4	RT=0
					WT=0 C=true
		$W_2(A)$			WT=2 C=false
					RT=0
Time	$R_1(A)$				
	Abort		$R_3(A)$		
			Delay		
		C			C=true
			$R_3(A)$		RT=3
				$W_4(A)$	

Basic Timestamps with Commit Bit

T$_1$	T$_2$	T$_3$	T$_4$	A
1	2	3	4	RT=0
				WT=0 C=true
	W$_2$(A)			WT=2 C=false
R$_1$(A)				RT=0
Abort		R$_3$(A)		
		Delay		
	C			C=true
		R$_3$(A)		RT=3
			W$_4$(A)	WT=4 C=false

Time

Basic Timestamps with Commit Bit

T$_1$	T$_2$	T$_3$	T$_4$	A
1	2	3	4	RT=0
				WT=0 C=true
	W$_2$(A)			WT=2 C=false
				RT=0
R$_1$(A)				
Abort		R$_3$(A)		
		Delay		
	C			C=true
		R$_3$(A)		RT=3
			W$_4$(A)	WT=4 C=false
		W$_3$(A)		

Time

Basic Timestamps with Commit Bit

T$_1$	T$_2$	T$_3$	T$_4$	A
1	2	3	4	RT=0
				WT=0 C=true
	W$_2$(A)			WT=2 C=false
R$_1$(A)				RT=0
Abort		R$_3$(A)		
		Delay		
	C			C=true
		R$_3$(A)		RT=3
			W$_4$(A)	WT=4 C=false
		W$_3$(A)		
		delay		

Time

Basic Timestamps with Commit Bit

T_1	T_2	T_3	T_4	A
1	2	3	4	RT=0
				WT=0 C=true
	$W_2(A)$			WT=2 C=false
				RT=0
$R_1(A)$				
Abort				
		$R_3(A)$		
		Delay		
	C			C=true
		$R_3(A)$		RT=3
			$W_4(A)$	WT=4 C=false
		$W_3(A)$		
		delay		
			abort	

Time

Basic Timestamps with Commit Bit

T$_1$	T$_2$	T$_3$	T$_4$	A
1	2	3	4	RT=0
				WT=0 C=true
	W$_2$(A)			WT=2 C=false
				RT=0
R$_1$(A)				
Abort				
		R$_3$(A)		
		Delay		
	C			C=true
		R$_3$(A)		RT=3
			W$_4$(A)	WT=4 C=false
		W$_3$(A)		
		delay		
			abort	WT=2 C=true

Time

Basic Timestamps with Commit Bit

T$_1$	T$_2$	T$_3$	T$_4$	A
1	2	3	4	RT=0
				WT=0 C=true
	W$_2$(A)			WT=2 C=false
R$_1$(A)				RT=0
Abort		R$_3$(A)		
		Delay		
	C			C=true
		R$_3$(A)		RT=3
			W$_4$(A)	WT=4 C=false
		W$_3$(A)		
		delay		
			abort	WT=2 C=true
		W3(A)		

Time

Basic Timestamps with Commit Bit

T_1	T_2	T_3	T_4	A
1	2	3	4	RT=0
				WT=0 C=true
	$W_2(A)$			WT=2 C=false
				RT=0
$R_1(A)$				
Abort				
		$R_3(A)$		
		Delay		
	C			C=true
		$R_3(A)$		RT=3
			$W_4(A)$	WT=4 C=false
		$W_3(A)$		
		delay		
			abort	WT=2 C=true
		W3(A)		WT=3 C=false

Time

Summary of Timestamp-based Scheduling

- View-serializable

- Avoids cascading aborts (hence: recoverable)

- Does NOT handle phantoms
  - These need to be handled separately, e.g. predicate locks

Multiversion Timestamp

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

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

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

Example (in class)

TS(T)=6

X_3 $\qquad$ X_9 $\qquad$ X_{12} $\qquad$ X_{18}

$R_6(X)$ -- what happens?
$W_{14}(X)$ – what happens?
$R_{15}(X)$ – what happens?
$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)

X_3 X_9 X_{12} X_{18}

$R_6(X)$ -- what happens? Return X_3

$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens?

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)

TS(T)=6

X_3 X_9 X_{12} X_{18}

$R_6(X)$ -- what happens? Return X_3
$W_{14}(X)$ – what happens?
$R_{15}(X)$ – what happens?
$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)

TS(T)=6

X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3
$W_{14}(X)$ – what happens?
$R_{15}(X)$ – what happens?
$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)

TS(T)=6 X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3
$W_{14}(X)$ – what happens?
$R_{15}(X)$ – what happens?
$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)

TS(T)=6

X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3
$W_{14}(X)$ – what happens?
$R_{15}(X)$ – what happens? Return X_{14}
$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)

$$X_3 \qquad X_9 \qquad X_{12} \quad X_{14} \quad X_{18}$$

$R_6(X)$ -- what happens? Return X_3

$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens? Return X_{14}

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)

TS(T)=6

$$X_3 \qquad X_9 \qquad X_{12} \quad X_{14} \quad X_{18}$$

$R_6(X)$ -- what happens? Return X_3

$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens? Return X_{14}

$W_5(X)$ – what happens? ABORT

When can we delete X_3?

Example (in class)

TS(T)=6

X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3
$W_{14}(X)$ – what happens?
$R_{15}(X)$ – what happens? Return X_{14}
$W_5(X)$ – what happens?   ABORT

When can we delete X_3?

Example (in class)

TS(T)=6

$$X_3 \qquad X_9 \qquad X_{12} \quad X_{14} \quad X_{18}$$

$R_6(X)$ -- what happens? Return X_3
$W_{14}(X)$ – what happens?
$R_{15}(X)$ – what happens? Return X_{14}
$W_5(X)$ – what happens? ABORT

When can we delete X_3? When min TS(T) > 9
  (i.e. all active transactions are later than time 9)

Details

- When $w_T(X)$ occurs,

  if the write is legal then

  create a new version, denoted X_t where $t = TS(T)$

Details

- When $w_T(X)$ occurs,

  if the write is legal then

  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 for that version
  - $RT(X_t)$ must still be maintained to check legality of writes

Details

- When $w_T(X)$ occurs,

   if the write is legal then

   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 for that version
   - $RT(X_t)$ must still be maintained to check legality of writes
        keep only the largest value

- Can delete X_t if we have a later version X_{t1} and all active transactions T have TS(T) > t1

Example w/ Basic Timestamps

	T_1	T_2	T_3	T_4	A
Timestamps:	1	2	3	4	RT=0
					WT=0
	$R_1(A)$				RT=1
	$W_1(A)$				WT=1
			$R_3(A)$		RT=3
			$W_3(A)$		WT=3
		$R_2(A)$			
		Abort			
				$R_4(A)$	RT=4

Example w/ Multiversion

T_1	T_2	T_3	T_4	A_0
1	2	3	4	
$R_1(A)$				RT=1

Example w/ Multiversion

T$_1$	T$_2$	T$_3$	T$_4$	A$_0$
1	2	3	4	
R$_1$(A) W$_1$(A)				RT=1

Example w/ Multiversion

T_1	T_2	T_3	T_4	A_0	A_1
1	2	3	4		
$R_1(A)$ $W_1(A)$				RT=1	Create

Example w/ Multiversion

T_1	T_2	T_3	T_4	A_0	A_1
1	2	3	4		
$R_1(A)$				RT=1	
$W_1(A)$					Create
		$R_3(A)$			

Example w/ Multiversion

T$_1$	T$_2$	T$_3$	T$_4$	A$_0$	A$_1$
1	2	3	4		
R$_1$(A)				RT=1	
W$_1$(A)					Create
		R$_3$(A)			RT=3

Example w/ Multiversion

T_1	T_2	T_3	T_4	A_0	A_1
1	2	3	4		
$R_1(A)$				RT=1	
$W_1(A)$					Create
		$R_3(A)$			RT=3
		$W_3(A)$			

Example w/ Multiversion

T_1	T_2	T_3	T_4	A_0	A_1	A_3
1	2	3	4			
$R_1(A)$				RT=1		
$W_1(A)$					Create	
		$R_3(A)$			RT=3	
		$W_3(A)$				Create

Example w/ Multiversion

T_1	T_2	T_3	T_4	A_0	A_1	A_3
1	2	3	4			
$R_1(A)$				RT=1		
$W_1(A)$					Create	
		$R_3(A)$			RT=3	
		$W_3(A)$				Create
	$R_2(A)$					

Example w/ Multiversion

T_1	T_2	T_3	T_4	A_0	A_1	A_3
1	2	3	4			
$R_1(A)$				RT=1		
$W_1(A)$					Create	
		$R_3(A)$			RT=3	
		$W_3(A)$				Create
	$R_2(A)$				RT=2	

Example w/ Multiversion

T_1	T_2	T_3	T_4	A_0	A_1	A_3
1	2	3	4			
$R_1(A)$				RT=1		
$W_1(A)$					Create	
		$R_3(A)$			RT=3	
		$W_3(A)$				Create
	$R_2(A)$				RT=2	

Keep only max RT

Example w/ Multiversion

T$_1$	T$_2$	T$_3$	T$_4$	A$_0$	A$_1$	A$_3$
1	2	3	4			
R$_1$(A)				RT=1		
W$_1$(A)					Create	
		R$_3$(A)			RT=3	
		W$_3$(A)				Create
	R$_2$(A)				RT=2	
	W$_2$(A)					

Example w/ Multiversion

T$_1$	T$_2$	T$_3$	T$_4$	A$_0$	A$_1$	A$_3$
1	2	3	4			
R$_1$(A)				RT=1		
W$_1$(A)					Create	
		R$_3$(A)			RT=3	
		W$_3$(A)				Create
	R$_2$(A)				~~RT=2~~	
	W$_2$(A)					
	abort					

Example w/ Multiversion

T$_1$	T$_2$	T$_3$	T$_4$	A$_0$	A$_1$	A$_3$
1	2	3	4			
R$_1$(A)				RT=1		
W$_1$(A)					Create	
		R$_3$(A)			RT=3	
		W$_3$(A)				Create
	R$_2$(A)				~~RT=2~~	
	W$_2$(A)					
	abort					
			R$_4$(A)			

Example w/ Multiversion

T_1	T_2	T_3	T_4	A_0	A_1	A_3
1	2	3	4			
$R_1(A)$ $W_1(A)$				RT=1	Create RT=3	
		$R_3(A)$ $W_3(A)$				Create
	$R_2(A)$ $W_2(A)$ **abort**				~~RT=2~~	
			$R_4(A)$			RT=4

Second Example w/ Multiversion

T_1	T_2	T_3	T_4	T_5	A_0	A_1	A_2	A_3	A_4	A_5
1	2	3	4	5						
			$W_4(A)$							

Second Example w/ Multiversion

T_1	T_2	T_3	T_4	T_5	A_0	A_1	A_2	A_3	A_4	A_5
1	2	3	4	5						
			$W_4(A)$						Create	
$W1(A)$						Create				
	$R_2(A)$					RT=2				
		$R_3(A)$				RT=3				
	$W_2(A)$									
	abort			$R_5(A)$					RT=5	
				$W_5(A)$						Create
			$R_4(A)$						RT=5	
$R_1(A)$						RT=3				
C					X					
		C			X					

X means that we can delete this version

Multiversion Concurrency Control

- Does **not** solve cascading aborts and recoverability, need to add Commit bit for those

- Does allow transactions to read in cases when they would have to abort in simple time-stamp based control
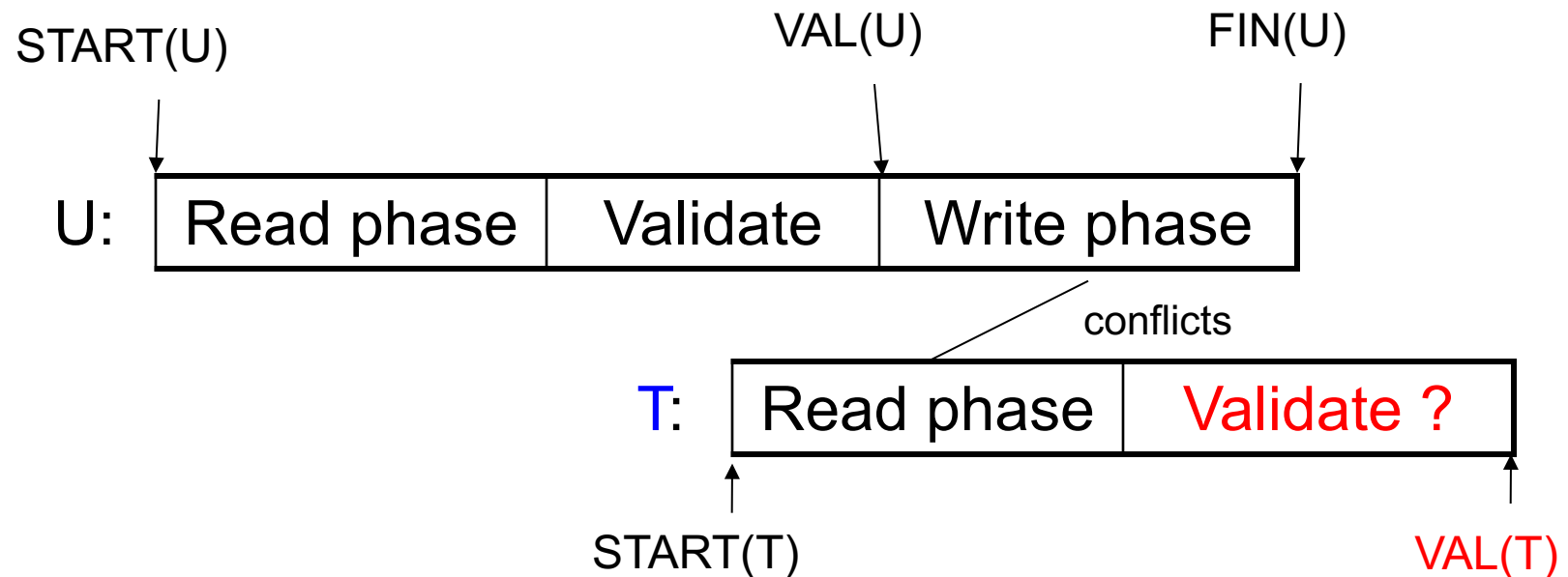
Outline

- Concurrency control by timestamps (18.8)
- Concurrency control by validation (18.9)
- Snapshot Isolation

Concurrency Control by Validation

- Each transaction T defines:
    - Read set RS(T) = the elements it reads
    - Write set WS(T) = the elements it writes

- Each transaction T has three phases:
    - Read phase; time = START(T)
    - Validate phase (may need to rollback); time = VAL(T)
    - Write phase; time = FIN(T)
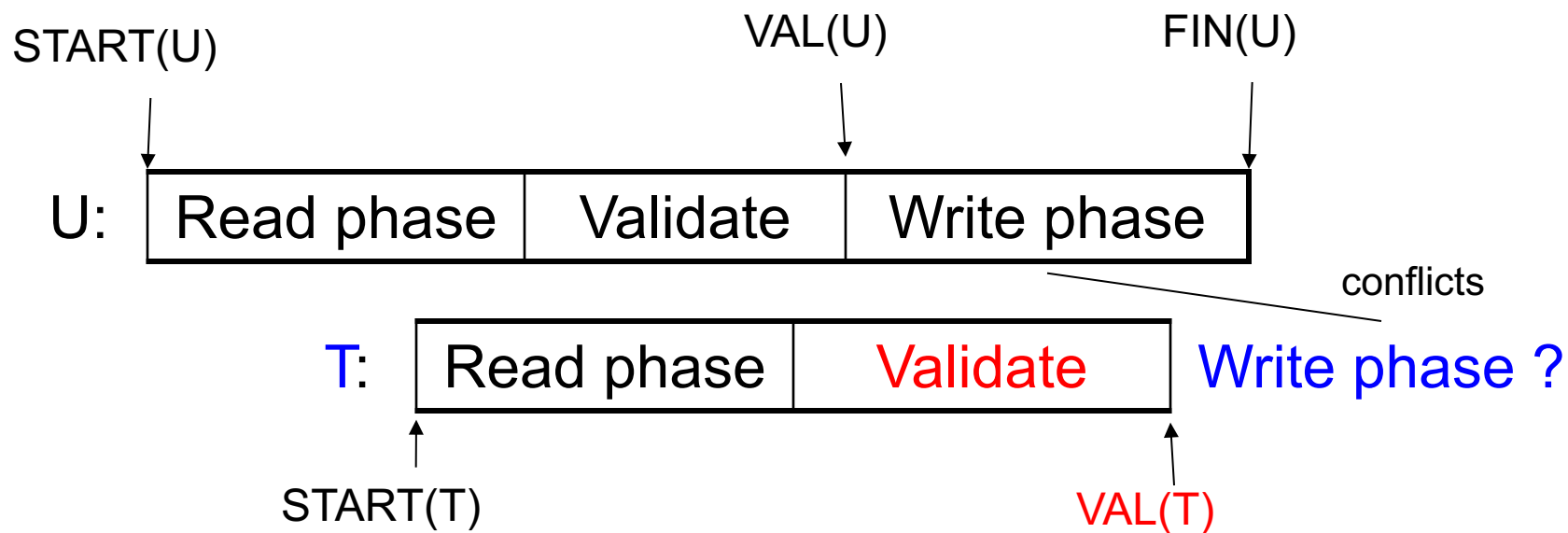
Main invariant: the serialization order is VAL(T)

Avoid $r_T(X)$ - $w_U(X)$ Conflicts

START(U)   VAL(U)   FIN(U)

U: | Read phase | Validate | Write phase |

                                    conflicts

T: | Read phase | Validate ? |

        START(T)   VAL(T)

IF RS(T) ∩ WS(U) and FIN(U) > START(T)
        (U has validated and U has not finished before T begun)
Then ROLLBACK(T)

Avoid $w_T(X) - w_U(X)$ Conflicts

START(U)   VAL(U)   FIN(U)

U: | Read phase | Validate | Write phase |

conflicts

T: | Read phase | Validate | Write phase ?

START(T)   VAL(T)

IF WS(T) ∩ WS(U) and FIN(U) > VAL(T)
     (U has validated and U has not finished before T validates)
Then ROLLBACK(T)

Outline

- Concurrency control by timestamps (18.8)
- Concurrency control by validation (18.9)
- Snapshot Isolation
  - Not in the book, but good(?) overview in Wikipedia

Snapshot Isolation

- A type of multiversion concurrency control algorithm
- Combines techniques we learned:
  - Timestamps
  - Multiversion
  - Validation

- Very popular: Oracle, PostgreSQL, SQL Server 2005
- Prevents many classical anomalies BUT...
      ...not serializable (!)
- "Serializable snapshot isolation" now in PostgreSQL

Snapshot Isolation Overview

- Each transactions receives a timestamp TS(T)

- Transaction T sees snapshot at time TS(T) of the database

- W/W conflicts resolved by "**first committer wins**" rule
  - Loser gets aborted

- R/W conflicts are ignored

Snapshot Isolation Details

- **Multiversion concurrency control:**
  - Versions of X: X_{t1}, X_{t2}, X_{t3}, . . .

- **When T reads X, return $X_{TS(T)}$.**

- **When T writes X (to avoid lost update):**
  - If latest version of X is TS(T) then proceed
  - Else if C(X) = true then abort
  - Else if C(X) = false then wait

- When T commits, write its updates to disk

What Works and What Not

- **Reads are ever delayed!**

- No dirty reads (Why ?)
  - Start each snapshot with consistent state

- No inconsistent reads (Why ?)
  - Two reads by the same transaction will read same snapshot

- No lost updates ("first committer wins")

- However: read-write conflicts not caught!
  - A txn can read and commit even though the value had changed in the middle

Write Skew

T1:
  READ(X);
  if X >= 50
      then Y = -50; WRITE(Y)
  COMMIT

T2:
  READ(Y);
  if Y >= 50
      then X = -50; WRITE(X)
  COMMIT

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

Starting with X=50,Y=50, we end with X=-50, Y=-50.
Non-serializable !!!

Discussion: Tradeoffs

- **Pessimistic CC: Locks**
  - Great when there are many conflicts
  - Poor when there are few conflicts

- **Optimistic CC: Timestamps, Validation, SI**
  - Poor when there are many conflicts (rollbacks)
  - Great when there are few conflicts

- **Compromise**
  - READ ONLY transactions $\rightarrow$ timestamps
  - READ/WRITE transactions $\rightarrow$ locks

Commercial Systems

Always check documentation!

- **DB2**: Strict 2PL

- **SQL Server**:
  - Strict 2PL for standard 4 levels of isolation
  - Multiversion concurrency control for snapshot isolation

- **PostgreSQL:** SI; recently: serializable SI (!)

- **Oracle:** SI