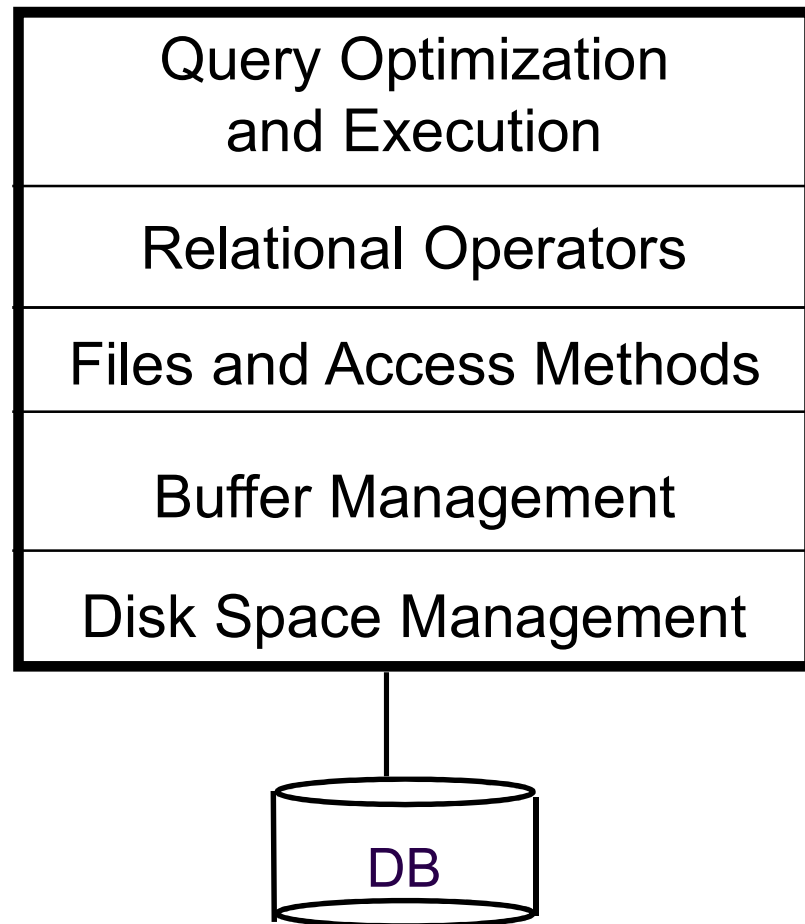


Transaction Management Overview

There are three side effects of acid.
Enhanced long term memory,
decreased short term memory, and
I forget the third.

- Timothy Leary

Structure of a DBMS



These layers must consider concurrency control and recovery
(Transaction, Lock, Recovery Managers)

Transactions

- Transaction (“xact”)- DBMS’s abstract view of a user program (or activity):
 - A **sequence** of **reads** and **writes** of database objects, e.g., a transaction that transfers \$100 from account A to account B can be expressed as:
 - Read Account A;
 - Write Updated Account A (\$100 less);
 - Read Account B;
 - Write Updated Account B (\$100 more);
 - **Unit of work** that must **commit** or **abort** as an **atomic unit**
- Transaction Manager controls the (correct) execution of Xacts
- User’s **program logic is invisible** to DBMS!
 - Arbitrary computation possible on data fetched from the DB
 - The DBMS only sees data read/written from/to the DB

ACID properties of Transaction Executions

- **Atomicity:** All actions in the Xact happen, or none happen
- **Consistency:** If each Xact is consistent, and the DB starts consistent, it ends up consistent
- **Isolation:** Execution of one Xact is isolated from that of other Xacts
- **Durability:** If a Xact **commits**, its effects persist

Atomicity and Durability

A.C.I.D.

- A transaction ends in one of two ways:
 - *commit* after completing all its actions
 - “commit” is a contract with the caller of the DB
 - *abort* (or be aborted by the DBMS) after executing some actions
 - Or *system crash* while the xact is in progress; treat as abort
- Two important properties for a transaction:
 - *Atomicity* : Either execute **all** its actions, or **none** of them
 - *Durability* : The effects of a committed xact **must** survive failures
- DBMS ensures the above by *logging* all actions
(Recovery):
 - *Undo* the actions of aborted/failed transactions
 - *Redo* actions of committed transactions not yet propagated to disk when system crashes

Transaction Consistency

A.C.I.D.

- Transactions preserve DB *consistency*
 - Given a consistent DB state, produce another consistent DB state
- DB Consistency expressed as a set of declarative **Integrity Constraints**
 - CREATE TABLE/ASSERTION statements
 - E.g. Each CS3223 student can only register in one project group. Each group must have 2 students
 - Application-level
 - E.g. Bank account total of each customer must stay the same during a “transfer” from savings to checking account
- Transactions that violate ICs are aborted
 - That’s all the DBMS can automatically check!

Isolation (Concurrency)

A.C.I.D.

- DBMS interleaves actions of many xacts concurrently
 - Actions = reads/writes of DB objects
- DBMS ensures xacts do not “step onto” one another
- Each xact executes as if it were running *by itself*
 - Concurrent accesses have no effect on a transaction’s behavior
 - Net effect *must be* identical to executing all transactions for *some serial order*
 - Users & programmers think about transactions in isolation
 - Without considering effects of other concurrent transactions!

Concurrency Control & Recovery

- Concurrency Control
 - Provide **correct** and **highly available** data access in the presence of **concurrent access** by many users
- Recovery
 - Ensures database is **fault tolerant**, and not corrupted by software, system or media failure
 - 24x7 access to mission critical data
- A boon to application developers!
 - Existence of CC&R allows applications to be written without explicit concern for concurrency and fault tolerance

Transactions

- A transaction (Xact) T_i can be viewed as a sequence of **actions**:
 - $r_i(O) = T_i$ reads an object O
 - $w_i(O) = T_i$ writes an object O
 - $c_i(O) = T_i$ completes successfully
 - $a_i(O) = T_i$ terminates unsuccessfully
- Each Xact must end with either a commit or an abort
- Example: A Xact T_1 that transfers 100 from A to B can be abstracted as

$T_1 : r_1(A), r_1(B), w_1(A), w_1(B), c_1$

Example:

T1: Read(A)
A \leftarrow A+100
Write(A)
Read(B)
B \leftarrow B+100
Write(B)

T2: Read(A)
A \leftarrow A \times 2
Write(A)
Read(B)
B \leftarrow B \times 2
Write(B)

Constraint: A=B

Schedule A: Serial Schedule


T1	T2	A	B
		25	25
Read(A); $A \leftarrow A+100$			
Write(A);		125	
Read(B); $B \leftarrow B+100$;			
Write(B);			125
	Read(A); $A \leftarrow A \times 2$;		
	Write(A);	250	
	Read(B); $B \leftarrow B \times 2$;		
	Write(B);		250
		250	250

Schedule B

		A	B
T1	T2	25	25
Read(A); $A \leftarrow A+100$			
Write(A);		125	
	Read(A); $A \leftarrow A \times 2$;		
	Write(A);	250	
Read(B); $B \leftarrow B+100$;			
Write(B);			125
	Read(B); $B \leftarrow B \times 2$;		
	Write(B);		250
		250	250

Schedule C

T1	T2	A	B
Read(A); $A \leftarrow A+100$		25	25
Write(A);		125	
	Read(A); $A \leftarrow A \times 2$;		
	Write(A);	250	
	Read(B); $B \leftarrow B \times 2$;		
	Write(B);		50
Read(B); $B \leftarrow B+100$;			150
Write(B);		250	150

Constraint not satisfied!! 

Schedule D

Same as Schedule C
but with new T2'

		A	B
T1	T2'	25	25
Read(A); $A \leftarrow A+100$			
Write(A);		125	
	Read(A); $A \leftarrow A \times 1$;		
	Write(A);	125	
	Read(B); $B \leftarrow B \times 1$;		
	Write(B);		25
Read(B); $B \leftarrow B+100$;			
Write(B);			125
		125	125

What are good schedules?

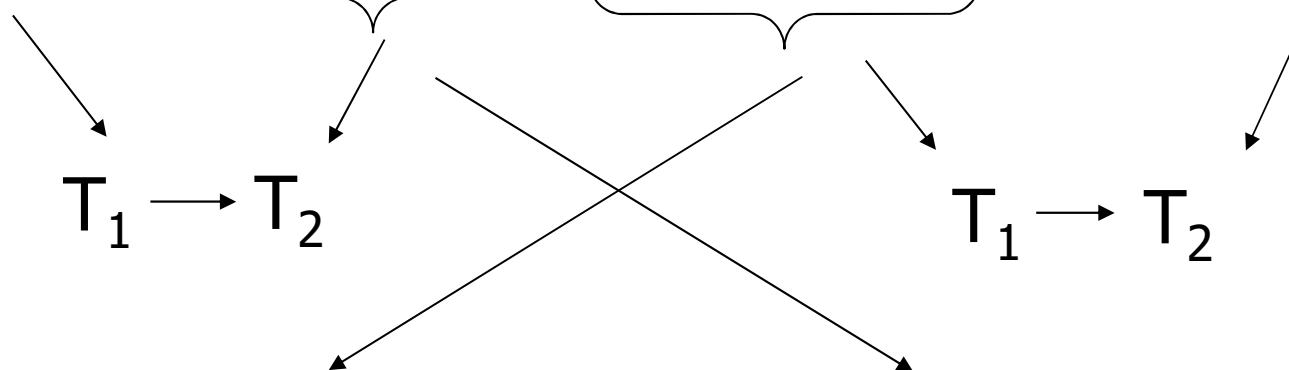
- Want schedules that are “good”, regardless of
 - initial state and
 - transaction semantics
- Only look at **order** of reads and writes

Example:

$$S_b = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$$

Example:

$S_b = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$




$S_{b'} = r_1(A)w_1(A) r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$

T_1 T_2

no cycles $\Rightarrow S_b$ is "equivalent" to a serial schedule $S_{b'}$ (in this case T_1, T_2)

Example (Cont)

Sd = $r_1(A)w_1(A)r_2(A)w_2(A)$ $r_2(B)w_2(B)$ $\underbrace{r_1(B)w_1(B)}$




Example (Cont)

$Sd = r_1(A)w_1(A)r_2(A)w_2(A) \ r_2(B)w_2(B)r_1(B)w_1(B)$

The diagram illustrates a sequence of operations: $r_1(A)w_1(A)r_2(A)w_2(A) \ r_2(B)w_2(B)r_1(B)w_1(B)$. Brackets group the operations for each variable: $r_1(A)w_1(A)$ and $r_1(B)w_1(B)$ are grouped together, as are $r_2(A)w_2(A)$ and $r_2(B)w_2(B)$. A curved arrow points from the first group to the second group, indicating a dependency. A red 'X' is drawn over the arrow, signifying a violation of a dependency constraint.

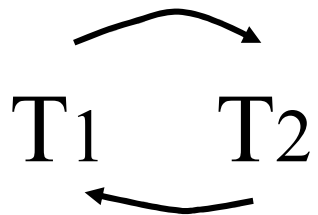
Example (Cont)

$S_d = r_1(A)w_1(A)r_2(A)w_2(A) \quad r_2(B)w_2(B)r_1(B)w_1(B)$



$T_1 \rightarrow T_2$

Also, $T_2 \rightarrow T_1$



S_d cannot be rearranged into a serial schedule

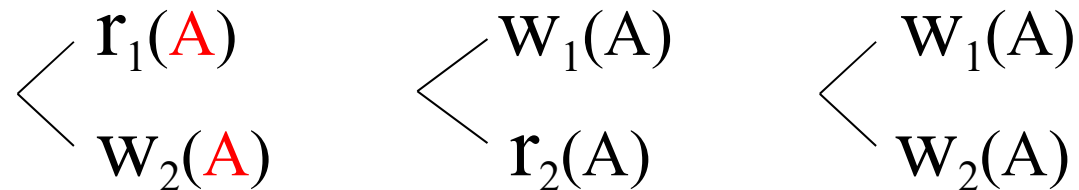
S_d is not “equivalent” to any serial schedule

S_d is “bad”

Concepts

Transaction: sequence of $r_i(x)$, $w_i(x)$ actions

Conflicting actions on the same item (A):



Schedule: represents *chronological order* in which *actions* of transactions are executed

Serial schedule: no interleaving of actions from different transactions

Serializable schedule: a schedule whose *effect* on *any consistent database instance* is guaranteed to be *identical* to that of *some* complete *serial schedule*

Anomalies with Interleaved Xact Executions

- Anomalies can arise due to conflicting actions
 - Dirty read problem (due to WR conflicts)
 - Unrepeatable read problem (due to RW conflicts)
 - Lost update problem (due to WW conflicts)

Dirty Read Problem: Example

- Dirty read problem (due to WR conflicts)
 - T_2 reads an object that has been modified by T_1 (which has not yet committed)
 - T_2 could see an inconsistent DB state!

T_1	T_2	Comments
		$x = 100$
R(x)		100
$x = x + 20$		
W(x)		$x = 120$
	R(x)	120
	$x = x \times 2$	
	W(x)	$x = 240$

- For every serial schedule, the final value of x is 200

Unrepeatable Read Problem: Example

- **Unrepeatable** read problem (due to RW conflicts)
 - T_2 **updates an object** that T_1 has just read while T_1 is still in progress
 - T_1 could get a different value if it reads the object again!

T_1	T_2	Comments
		$x = 100$
R(x)	R(x)	100
	$x = x - 20$	100
	W(x)	$x = 80$
R(x)		80

- For every serial schedule, both values read by T_1 are the same

Lost Update Problem: Example

- Lost update problem (due to WW conflicts)
 - T_2 overwrites the value of an object that has been modified by T_1 while T_1 is still in progress
 - T_1 's update is lost!

T_1	T_2	Comments
		$x = 100$
R(x)		100
	R(x)	100
$x = x + 20$		
	$x = x \times 2$	
W(x)		$x = 120$
	W(x)	$x = 200$

- For schedule (T_1, T_2) , the final value of x is 240
- For schedule (T_2, T_1) , the final value of x is 220

Up to this point ...

- We want actions of transactions to interleave (for better throughput)
- Interleaving of actions can mess up the entire database!
- We want the interleaved actions to be serializable

Definition

- S1, S2 are *conflict equivalent* schedules if S1 can be transformed into S2 by a series of swaps on *non-conflicting actions*
 - S1 and S2 order every pair of conflicting actions of two **committed** Xacts in the same way
- A schedule is *conflict serializable* if it is conflict equivalent to some serial schedule

Note: (a) Some serializable schedules are NOT conflict serializable.
A price we pay to achieve efficient enforcement.
(b) There are alternative (weaker) notions of serializability.

Conflict Equivalent: Example

S ₁	S ₂
$R_1(x)$	$R_2(x)$
$W_1(x)$	$W_2(y)$
$R_1(y)$	$R_1(x)$
$R_2(x)$	$W_1(x)$
$W_2(y)$	$R_1(y)$
$W_1(z)$	$W_1(z)$

Not conflict equivalent

S ₁	S ₃
$R_1(x)$	$R_1(x)$
$W_1(x)$	$W_1(x)$
$R_1(y)$	$R_1(y)$
$R_2(x)$	$W_1(z)$
$W_2(y)$	$R_2(x)$
$W_1(z)$	$W_2(y)$

Conflict equivalent

Conflict-Serializability is NOT necessary for Serializability

- S1: w1(Y); w1(X); w2(Y); w2(X); w3(X)
 - Serial schedule
- S2: w1(Y); w2(Y); w2(X); w1(X); w3(X)
 - Serializable?
- S1, S2 conflict equivalent?
- What is the problem?
 - In the schedule, what we essentially have are **blind writes**, i.e., a write on an object O that did not read O prior to the write

Precedence (Conflict Serializability)

graph $P(S)$ (S is schedule)

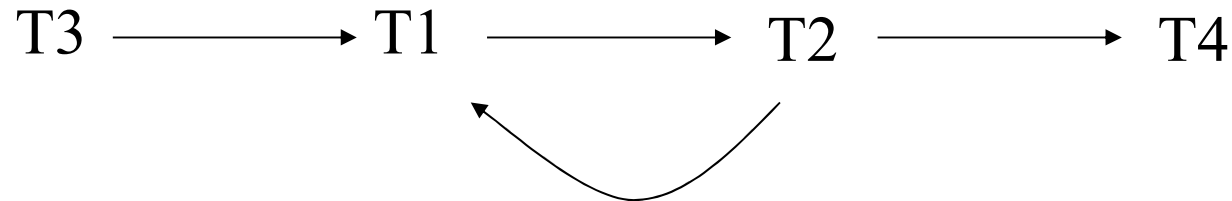
Nodes: transactions in S

Arcs: $T_i \rightarrow T_j$ whenever

- $p_i(A), q_j(A)$ are actions in S
- $p_i(A) <_S q_j(A)$
- at least one of p_i, q_j is a write

Exercise:

- What is $P(S)$ for
 $S = w_3(A) \ w_2(C) \ r_1(A) \ w_1(B) \ r_1(C) \ w_2(A) \ r_4(A) \ w_4(D)$



- Is S conflict serializable?

Theorem

$P(S)$ is acyclic $\Leftrightarrow S$ is conflict serializable

S_1, S_2 conflict equivalent $\Rightarrow P(S_1)=P(S_2)$???

$P(S_1)=P(S_2) \not\Rightarrow S_1, S_2$ conflict equivalent ???

$P(S_1)=P(S_2) \not\Rightarrow S_1, S_2 \text{ conflict equivalent}$

- $S_1 = w_3(A) \ w_2(C) \ r_1(C) \ r_1(A) \ w_2(B) \ w_1(B) \ w_2(A)$
- $S_2 = w_3(A) \ r_1(A) \ w_2(B) \ w_1(B) \ r_1(C) \ w_2(C) \ w_2(A)$

View Serializable Schedules

- Two schedules S and S' are **view equivalent** if they satisfy all the following conditions:
 - If T_i reads the initial value of A in S , then T_i must also read the initial value of A in S'
 - If T_i reads a value of A written by T_j in S , then T_i must also read the value of A written by T_j in S'
 - For each data object A , the Xact (if any) that performs the final write on A in S must also perform the final write on A in S'
- A schedule S is a **view serializable schedule** if S is view equivalent to some serial schedule over the same set of Xacts

View Equivalent Schedule: Example

S	S ₁
$R_1(x)$	$R_1(x)$
$R_2(y)$	$R_1(y)$
$W_3(x)$	$W_1(z)$
$W_3(z)$	$R_2(y)$
$R_2(x)$	$R_2(x)$
$R_1(y)$	$W_2(z)$
$W_1(z)$	$W_3(x)$
$W_2(z)$	$W_3(z)$

Not view equivalent

S	S ₂
$R_1(x)$	$R_1(x)$
$R_2(y)$	$R_1(y)$
$W_3(x)$	$W_1(z)$
$W_3(z)$	$W_3(x)$
$R_2(x)$	$W_3(z)$
$R_1(y)$	$R_2(y)$
$W_1(z)$	$R_2(x)$
$W_2(z)$	$W_2(z)$

View equivalent/serializable

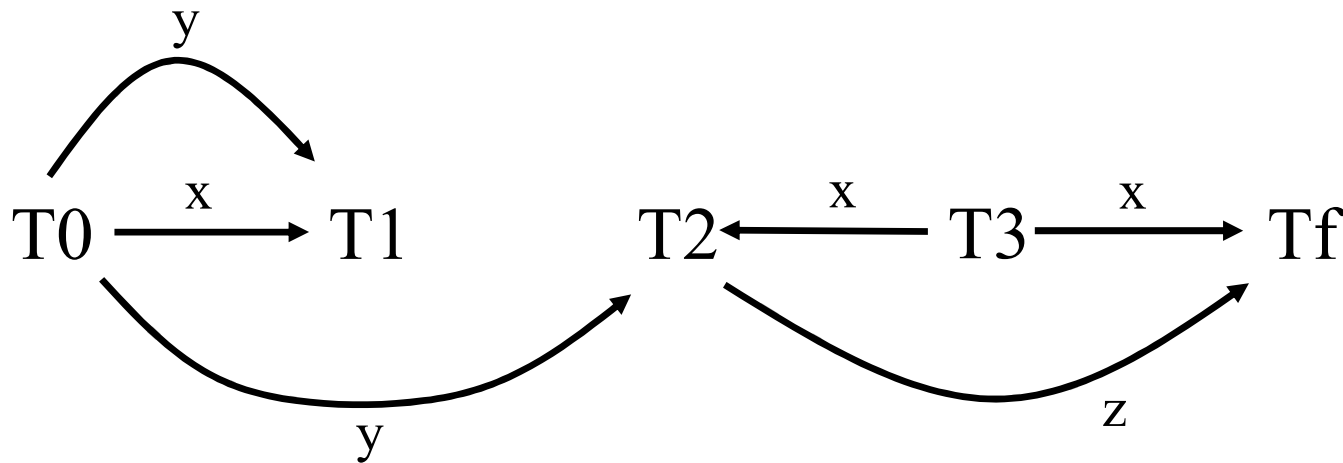
Test for View Serializability

- Polygraphs
 - Generalization of precedence graphs for testing view serializability
- Theorem: $\text{Polygraph}(S)$ is acyclic $\Leftrightarrow S$ is view serializable

Polygraph of a schedule S

- 3 rules
 - Nodes
 - A node for each transaction
 - An additional node representing a hypothetical transaction T_0 that wrote initial values for each item read by any transaction in the schedules
 - An additional node representing a hypothetical transaction T_f that reads every item written by one or more transactions after each schedule ends
 - Edges
 - For each action $r_i(X)$ with source T_j , add an arc from T_j to T_i

Polygraph of S: Rules 1 and 2



S
$R_1(x)$
$R_2(y)$
$W_3(x)$
$W_3(z)$
$R_2(x)$
$R_1(y)$
$W_1(z)$
$W_2(z)$

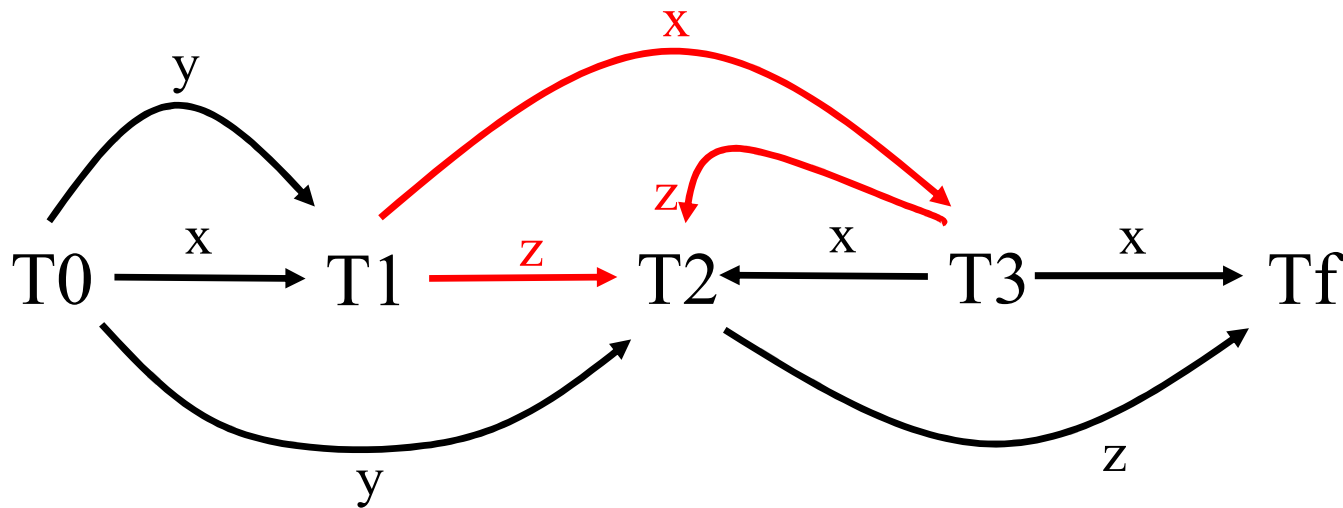
Rule 3

- Let T_j be the source of a read $r_i(X)$, i.e., $T_j \rightarrow T_i$
- Let T_k be another transaction that also writes X
- We cannot allow T_k to intervene between T_j and T_i
- So, T_k must be before T_j or after T_i
- Add edges $T_k \rightarrow T_j$ and $T_i \rightarrow T_k$ to the polygraph
 - Intuitively, only one of these edges is “real”; and we can choose either of them when we try to make the polygraph acyclic at the end of the process
 - Special cases:
 - If T_j is T_0 then it is not possible for T_k to appear before T_j so we need only to add one edge $T_i \rightarrow T_k$
 - If T_i is T_f then it is not possible for T_k to appear after T_i so we need only to add one edge $T_k \rightarrow T_j$

Simplifications for rule 3

- When avoiding interference with the edge $T_j \rightarrow T_i$, only need to consider T_k that are
 - Writers of an item that caused the edge $T_j \rightarrow T_i$
- No need to consider T_0 or T_f (since they can never be T_k)
- No need to consider T_i or T_j , which are the ends of the edge itself

Complete Polygraph of S: Rules 1 – 3



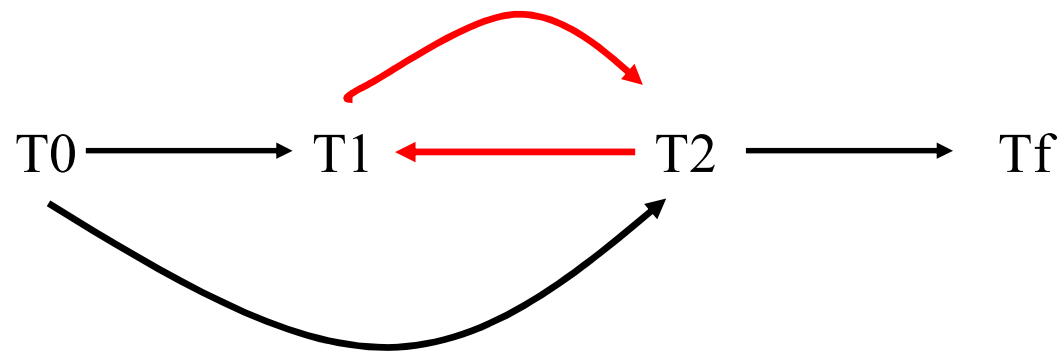
S
$R_1(x)$
$R_2(y)$
$W_3(x)$
$W_3(z)$
$R_2(x)$
$R_1(y)$
$W_1(z)$
$W_2(z)$

The graph is acyclic – with the serial order of T1; T3; T2

It is view serializable!

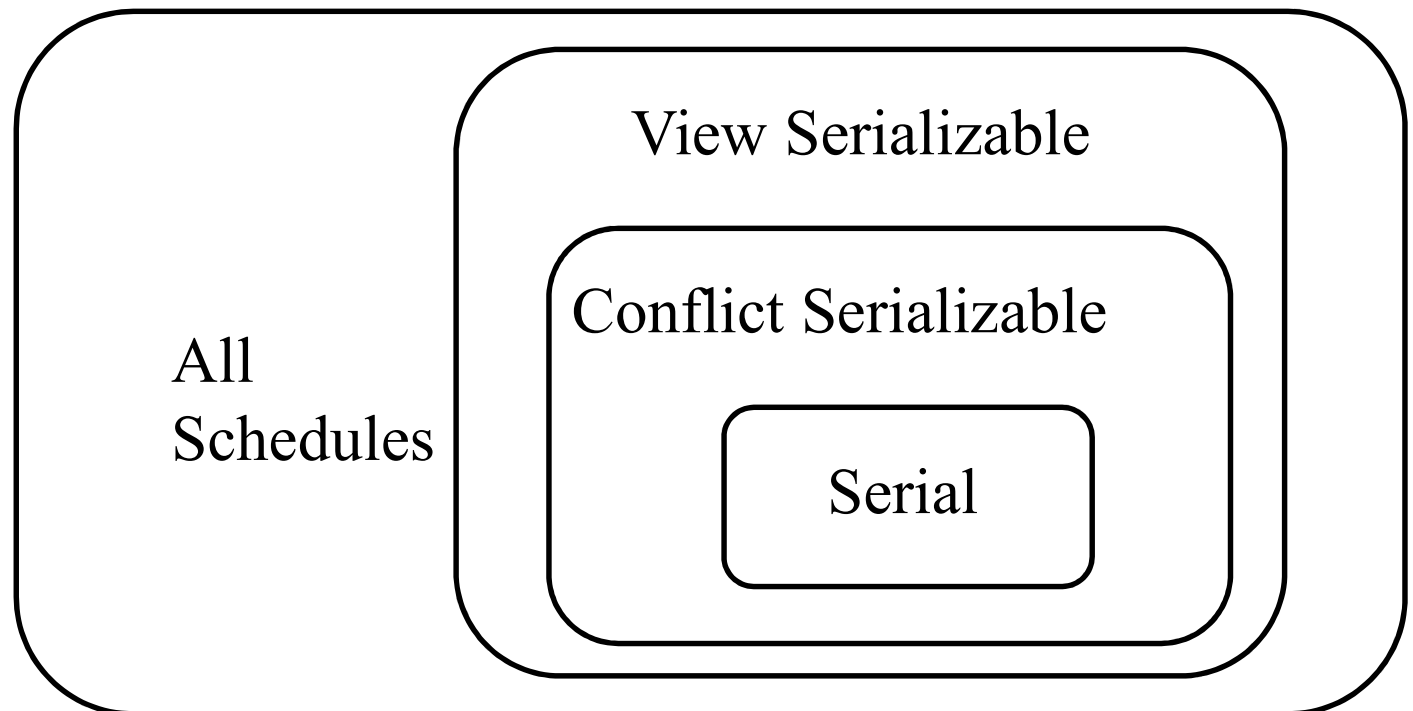
Is the following schedule view serializable?

- $R1(X), R2(X), W1(X), W2(X)$



Conflict vs View Serializability

- Theorem: A schedule that is conflict serializable is also view serializable
- View Serializability: Allows all conflict serializable schedules + blind writes!
- Theorem: If S is view serializable and S has no blind writes, then S is also conflict serializable



Cascading Aborts

- For correctness, if T_i has read from T_j , then T_i must abort if T_j aborts

T_1	T_2
$W_1(x)$	$R_2(x)$ $W_2(y)$

- T_1 's abort is cascaded to T_2
- Recursive aborting process is known as **cascading aborts**

Recoverable Schedules

- A schedule S is said to be a **recoverable schedule** if for every Xact T that commits in S, T must commit after T' if T reads from T'

T_1	T_2
$W_1(x)$	$R_2(x)$
	$W_2(y)$
	<i>Commit₂</i>

Cascadeless (Avoid Cascading Aborts) Schedules

- While recoverable schedules guarantee that committed Xacts will not be aborted, cascading aborts of active Xacts are possible
 - **Example:** if T_i reads from T_j and T_j aborts, T_i must also abort
- Cascading aborts are undesirable because of the cost of bookkeeping to identify them and the performance penalty incurred
- To avoid cascading aborts (or to be cascadeless), DBMS must permit **reads only from committed Xacts**
- A schedule S is a **cascadeless schedule** if whenever T_i reads from T_j in S , Commit_j must precede this read action
- **Theorem 4:** A cascadeless schedule is also a recoverable schedule

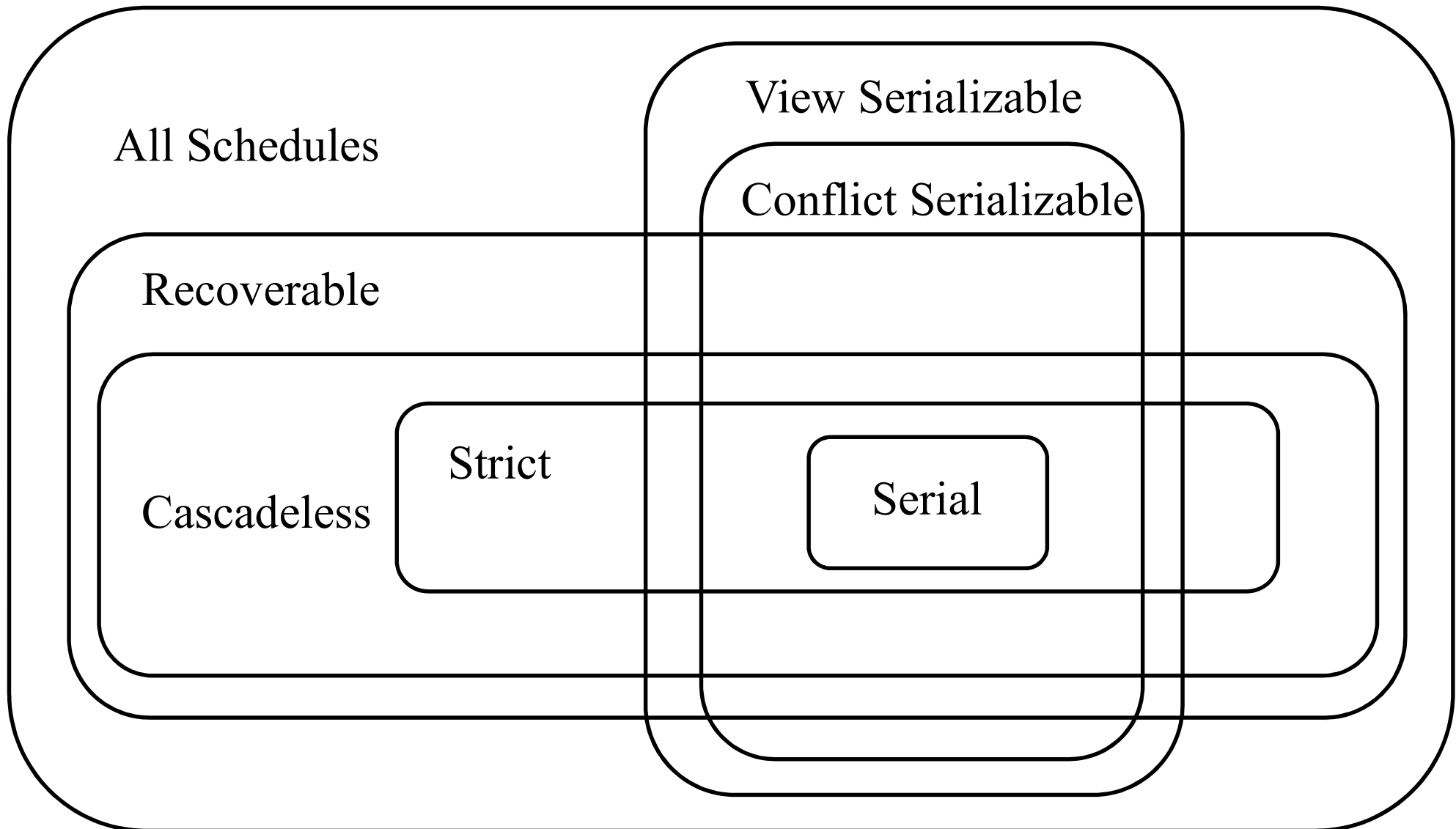
Strict Schedules

- A schedule S is a **strict schedule** if for every $W_i(O)$ in S , O is **not read or written** by another X act until T_i either aborts or commits
- **Theorem 5:** A strict schedule is also a cascadeless schedule

Cascadeless But Not Strict Schedules

T1	T2	T1	T2
R(A)	R(A)	W(A)	W(A)
W(A)			Commit
	W(A)	R(A)	
Abort			
	Commit		

Relationships between schedules



Concurrency Control

Smile, it is the key that fits the
lock of everybody's heart.

Anthony J. D'Angelo,
The College Blue Book

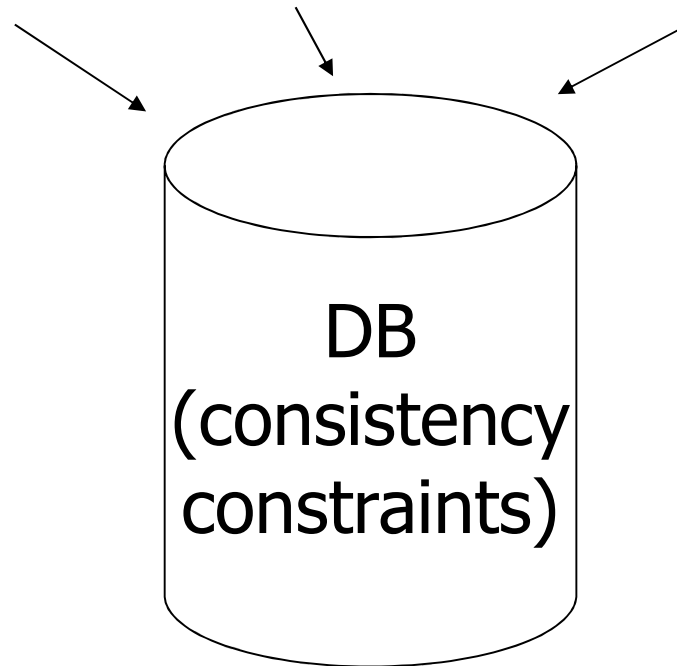
Concurrency Control

T1

T2

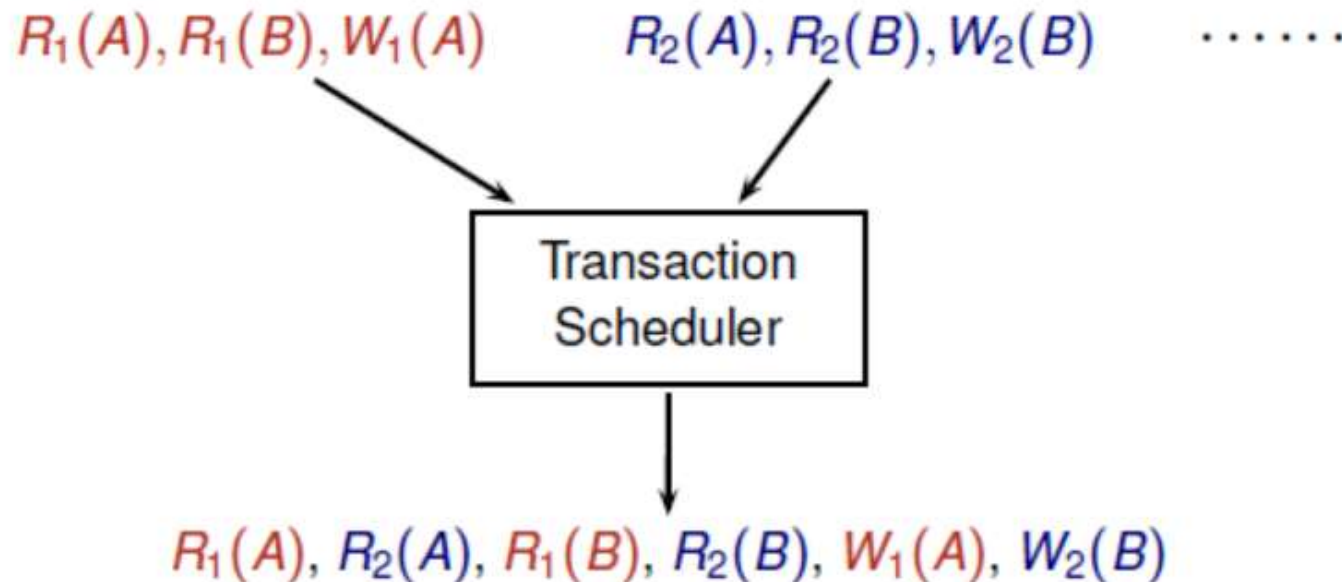
...

T_n



Improves
latency and
throughput

How to enforce serializable schedules?



- For each input action (read, write, commit, abort) to the scheduler, the scheduler performs one of the followings:
 - Output the action to the schedule
 - Postpone the action by blocking the transaction, or
 - Reject the action and abort the transaction

Concurrency Control (CC) Algorithms

- Pessimistic CC
 - Lock-based CC
 - Timestamp-based CC
- Multiversion CC
- Optimistic CC

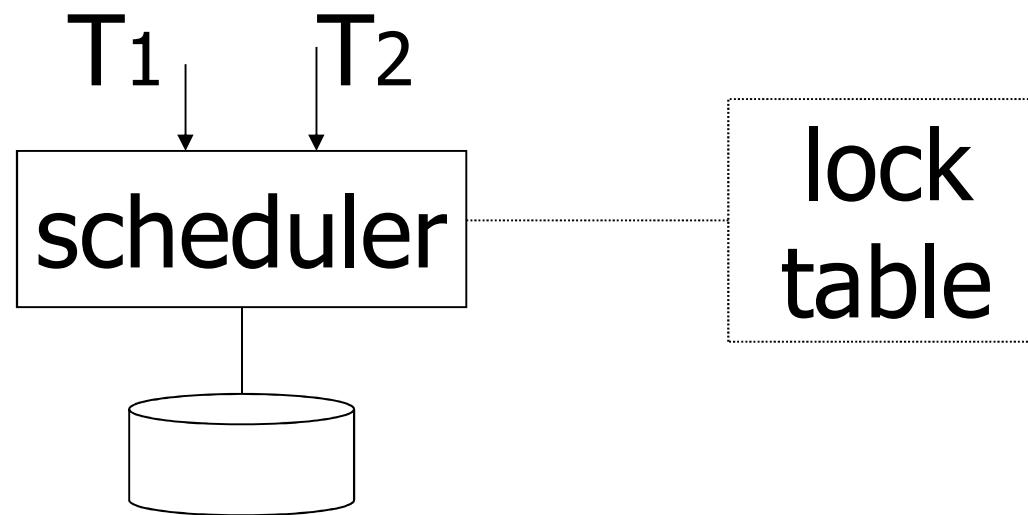
Pessimistic Concurrency Control

A locking protocol

Two new actions:

lock (exclusive): $li(A)$

unlock: $ui(A)$

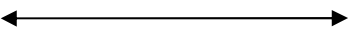


Rules

Rule #1: Well-formed transactions

$T_i: \dots li(A) \dots pi(A) \dots ui(A) \dots$

Rule #2: Legal scheduler

$S = \dots li(A) \dots ui(A) \dots$

no $lj(A)$

Exercise:

- **What schedules are legal?**
What transactions are well-formed?

$S1 = l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B)$
 $r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

Not legal

$S2 = l_1(A)r_1(A)w_1(B)u_1(A)u_1(B)$
 $l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$

Not well-formed

Not legal

$S3 = l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)$
 $l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

Schedule F (Schedule C with locking)

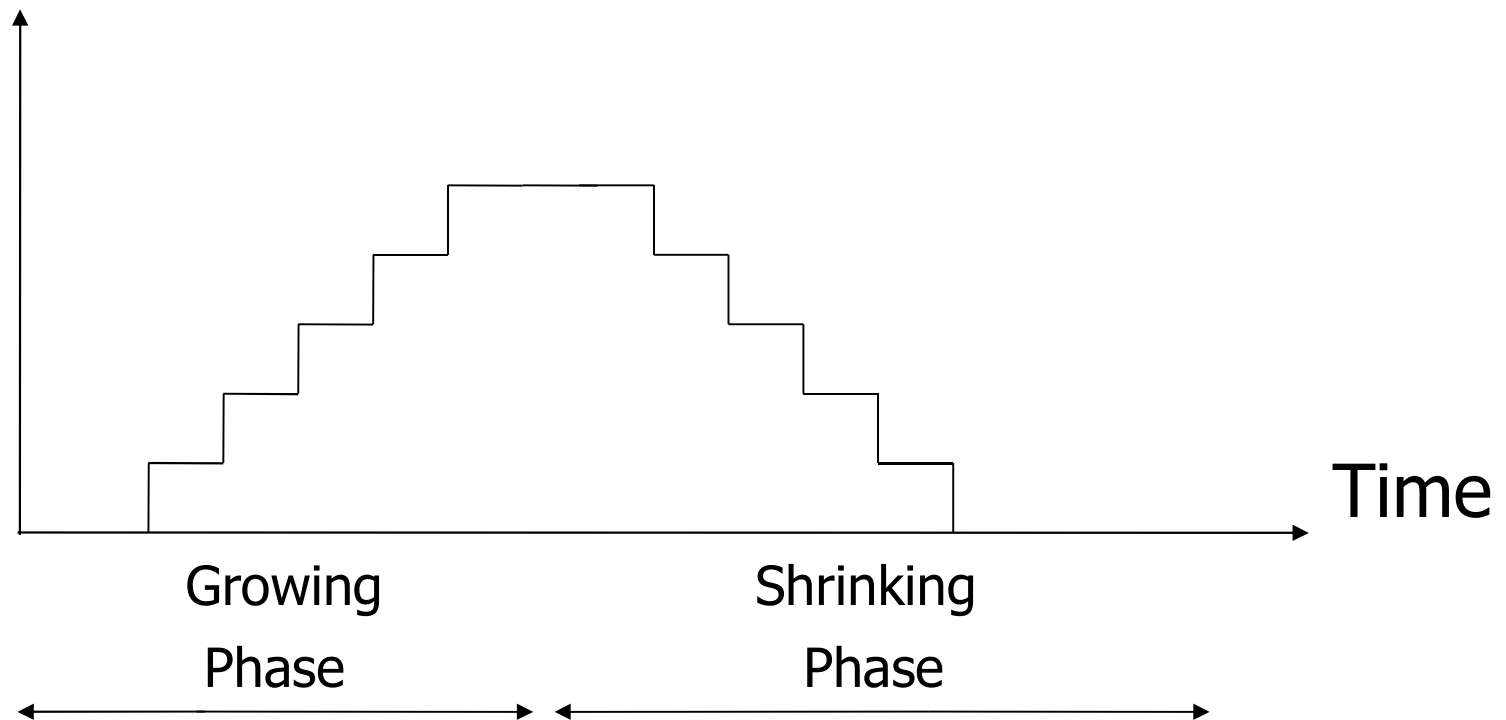
		A	B
T1	T2	25	25
$l_1(A); \text{Read}(A)$			
$A \leftarrow A + 100; \text{Write}(A); u_1(A)$		125	
	$l_2(A); \text{Read}(A)$		
Rules 1 & 2 are not enough!		250	
	$l_2(B); \text{Read}(B)$		
	$B \leftarrow B \times 2; \text{Write}(B); u_2(B)$		50
$l_1(B); \text{Read}(B)$			
$B \leftarrow B + 100; \text{Write}(B); u_1(B)$			150
		250	150

Rule #3 Two phase locking (2PL) for transactions

$T_i = \dots \dots \text{li}(A) \dots \dots \dots \text{ui}(A) \dots \dots \dots$

← no unlocks | | no locks →

locks
held by
 T_i



Schedule G

T1	T2
----	----

$l_1(A); \text{Read}(A)$

$A \leftarrow A+100; \text{Write}(A)$

$l_1(B); u_1(A)$

$\text{Read}(B); B \leftarrow B+100$

$\text{Write}(B); u_1(B)$

$l_2(A); \text{Read}(A)$

$A \leftarrow A \times 2; \text{Write}(A);$

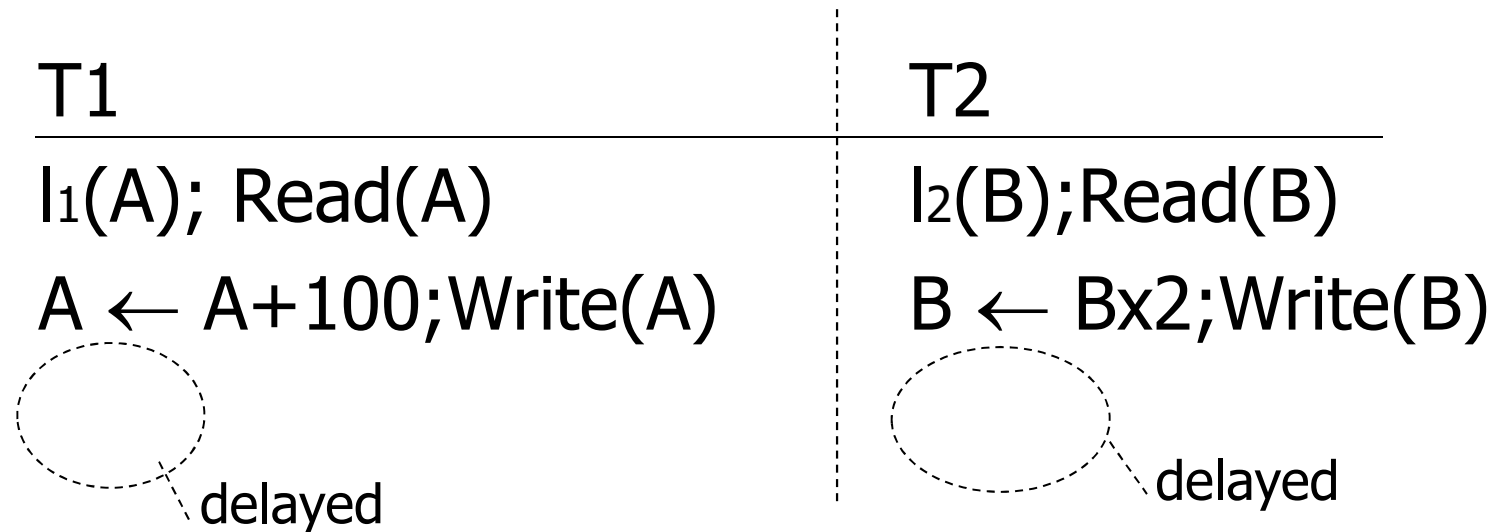
delayed

$l_2(B); u_2(A); \text{Read}(B)$

$B \leftarrow B \times 2; \text{Write}(B); u_2(B);$

T1	T2
$l_1(A); \text{Read}(A)$	$l_2(A); \text{Read}(A)$
$A \leftarrow A+100; \text{Write}(A); u_1(A)$	$A \leftarrow A \times 2; \text{Write}(A); u_2(A)$
	$l_2(B); \text{Read}(B)$
	$B \leftarrow B \times 2; \text{Write}(B); u_2(B)$
$l_1(B); \text{Read}(B)$	
$B \leftarrow B+100; \text{Write}(B); u_1(B)$	

Schedule H (*T2 reversed*)



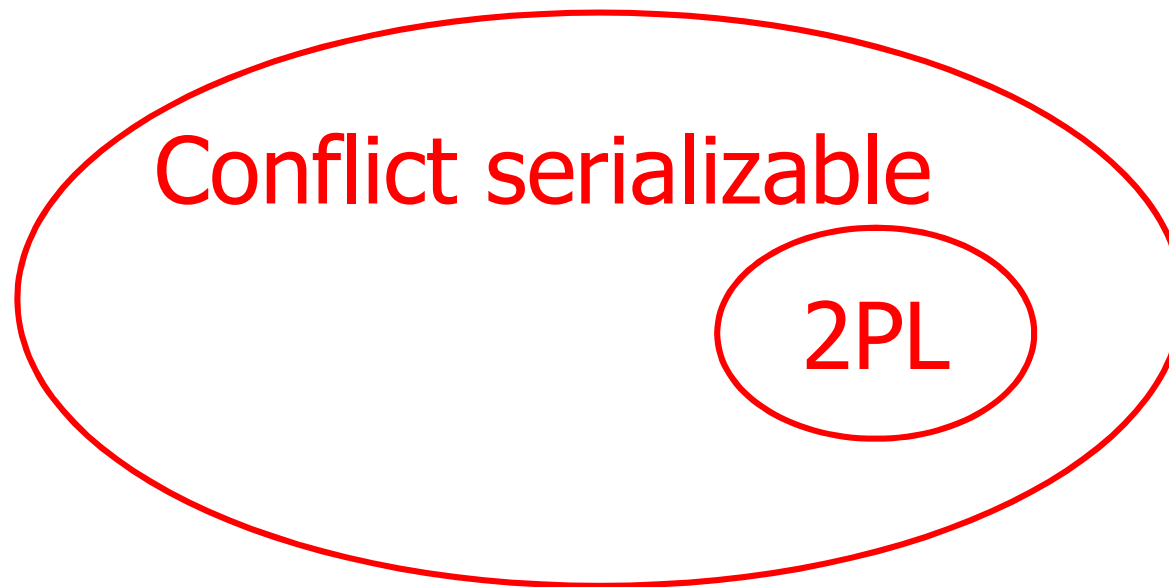
Transactions are *deadlocked*

- Some deadlocked transactions are *aborted* and *rolled back* (and all their actions undone)

Theorem

Rules #1,2,3 (2PL) \Rightarrow conflict serializable
schedule

2PL subset of conflict serializable



Are the following schedules conflict serializable and can be produced by 2PL?

$S_1 : R_1(B)W_3(A)R_2(C)W_2(C)R_3(C)W_1(B)R_3(B)W_3(C)$

$S_2 : R_1(A)W_1(A)R_2(B)W_2(C)R_2(A)R_3(C)W_3(B)R_1(B)$

What else?

- Beyond this simple 2PL protocol, it is all a matter of improving performance and allowing more concurrency....
 - Shared locks
 - Multiple granularity
 - Inserts, deletes and phantoms
 - **Other types of CC mechanisms**

Shared locks

So far (exclusive lock):

S1 = ...l1(A) **r1(A)** u1(A) ... l2(A) **r2(A)** u2(A) ...



Do not conflict (but executed serially!)

Instead:

S2 = ... ls1(A) r1(A) **ls2(A) r2(A)** us1(A) **us2(A)**



Interleaved operations

- Both transactions can run concurrently
- Better performance

Lock actions

$l\text{-}t_i(A)$: lock A in t mode (t is S or X)

$u\text{-}t_i(A)$: unlock t mode (t is S or X)

Shorthand:

$u_i(A)$: unlock whatever modes T_i has locked A

Rule #1 Well formed transactions

$T_i = \dots l-S_1(A) \dots r_1(A) \dots u_1(A) \dots$

$T_i = \dots l-X_1(A) \dots w_1(A) \dots u_1(A) \dots$

What about transactions that read and write the same object?

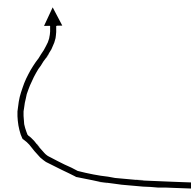
Option 1: Request exclusive lock

$T_i = \dots \textcolor{red}{1-X1(A)} \dots r1(A) \dots w1(A) \dots u(A) \dots$

Option 2: Upgrade

(E.g., need to read, but don't know if will write...)

$T_i = \dots \textcolor{red}{1-S1(A)} \dots r1(A) \dots \textcolor{red}{1-X1(A)} \dots w1(A) \dots u(A) \dots$

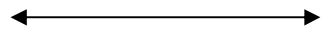


Think of

- Get 2nd lock on A, or
- Drop S, get X lock

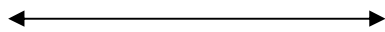
Rule #2 Legal scheduler

$S = \dots l-S_i(A) \dots \dots u_i(A) \dots$



no $l-X_j(A)$

$S = \dots l-X_i(A) \dots \dots u_i(A) \dots$



no $l-X_j(A)$

no $l-S_j(A)$

A way to summarize Rule #2

Compatibility matrix

New request

Lock
already
held in

		New request	
		S	X
Lock already held in	S	True	False
	X	False	False

Rule # 3 2PL transactions

No change except for upgrades:

- (I) If upgrade gets more locks
(e.g., $S \rightarrow \{S, X\}$) then no change!
- (II) If upgrade releases read (shared)
lock (e.g., $S \rightarrow X$)
 - can be allowed in growing phase

Theorem

Rules 1,2,3 \Rightarrow Conf.serializable
for S/X locks schedules

Example

T1

$l-S_1(A); r_1(A)$

$l-X_1(B)$ (Denied)

$l-X_1(B); r_1(B); w_1(B)$

$u_2(A); u_2(B)$

T2

$l-S_2(A); r_2(A)$

$l-S_2(B); r_2(B)$

$u_2(A); u_2(B)$

Strict 2PL Protocol

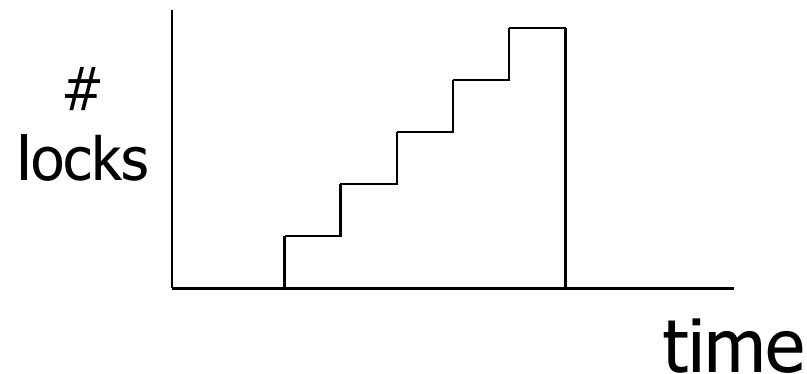
- Same as 2PL except a Xact must hold on to locks until Xact commits or aborts
- Theorem: Strict 2PL schedules are strict and conflict serializable

How does locking work in practice?

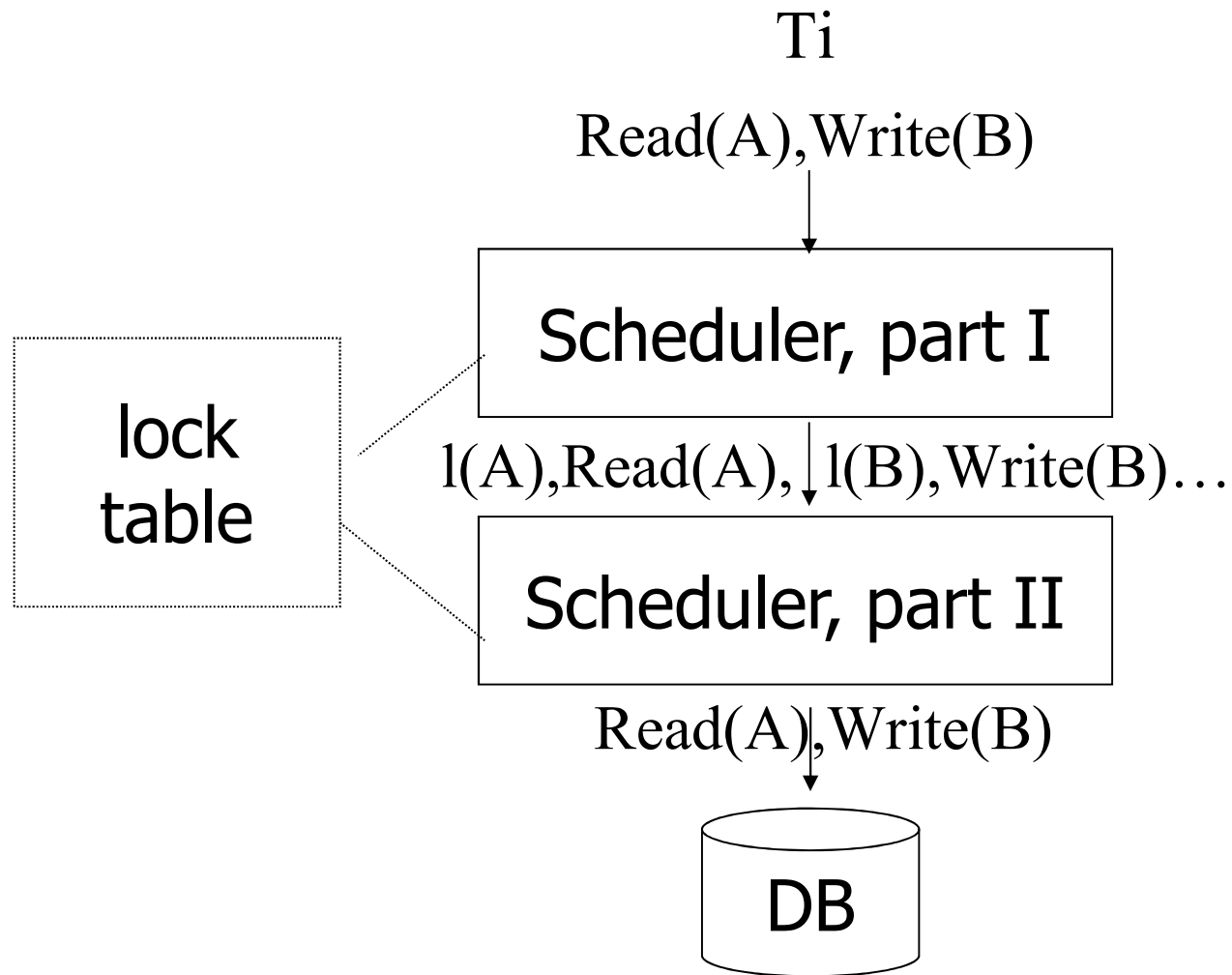
- Every system is different
(E.g., may not even provide
CONFLICT-SERIALIZABLE schedules)
- But here is one (simplified) way ...

Sample Locking System:

- (1) Don't trust transactions to request/release locks
- (2) Hold all locks until transaction commits

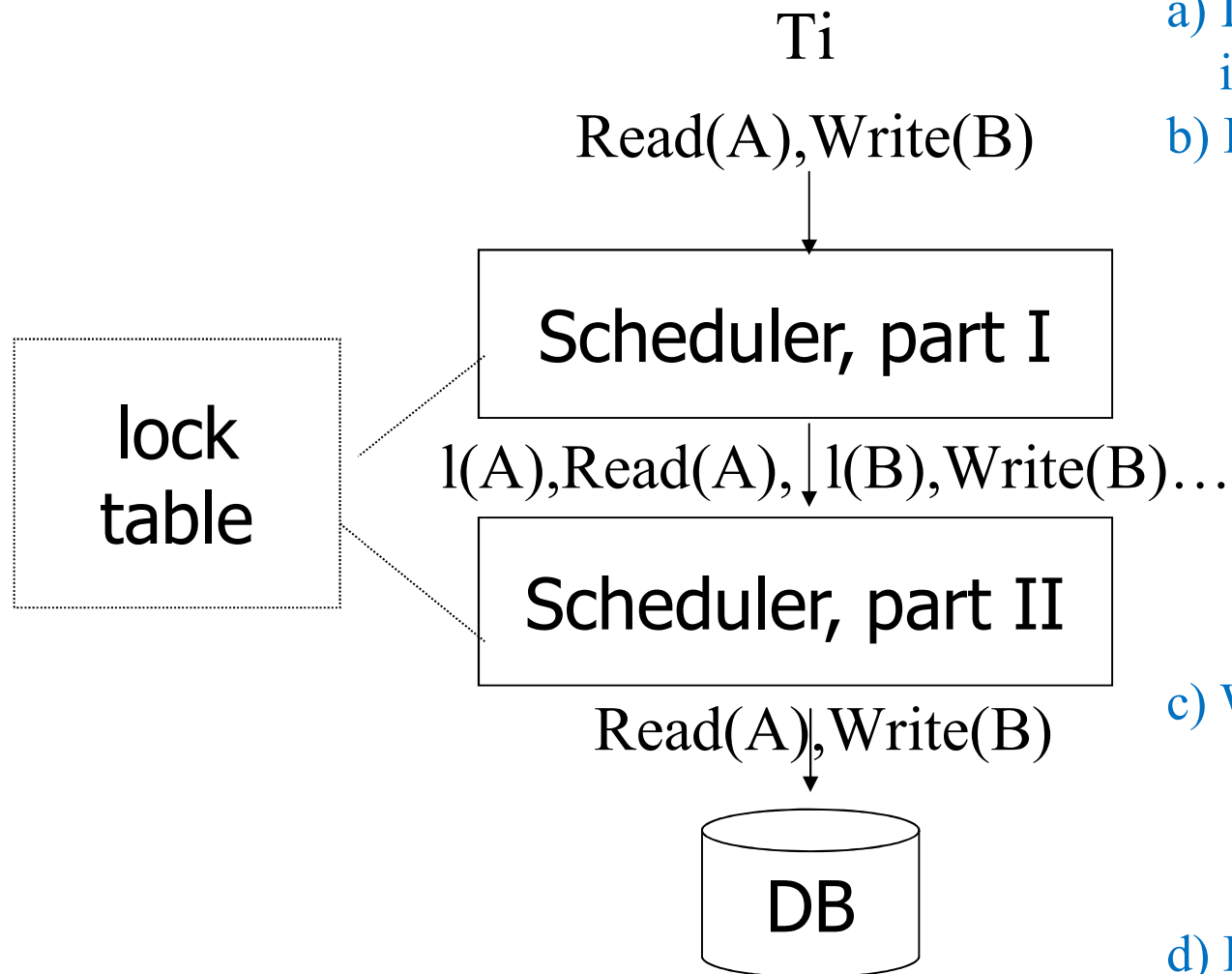


Architecture of a Locking Scheduler



Part I: select appropriate lock mode, and inserts appropriate lock actions ahead of all database operations

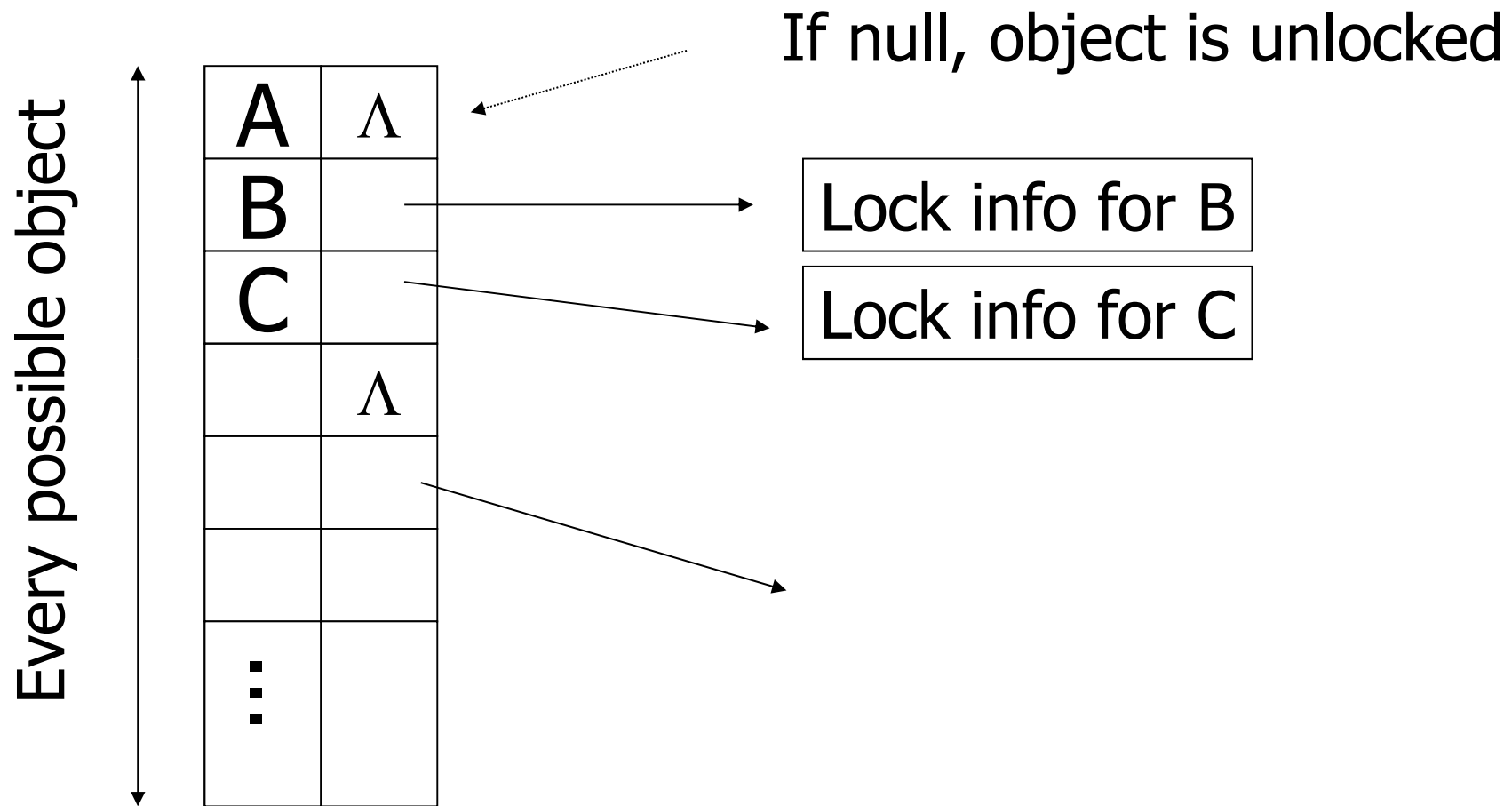
Architecture of a Locking Scheduler



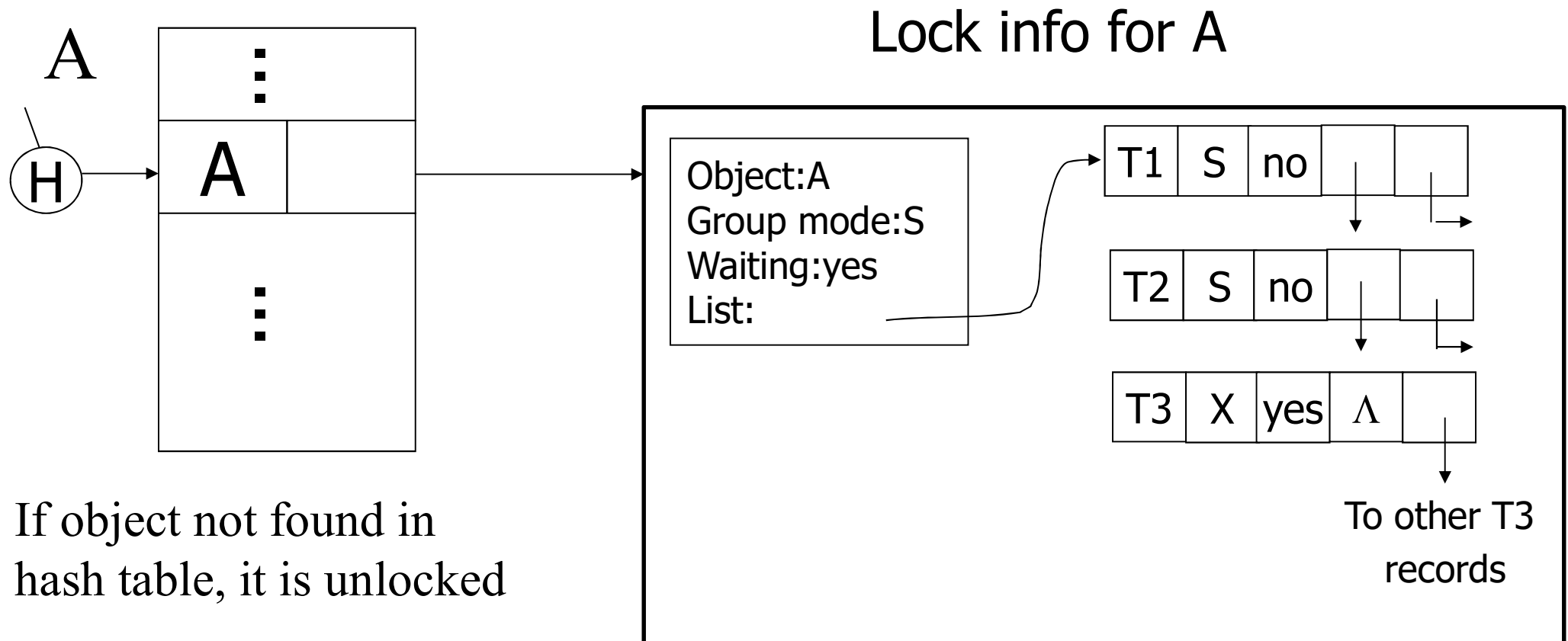
Part II: Execute the operations

- It determines if lock should be granted; if not, then transaction is delayed.
- If transaction is not delayed,
 - If action is a normal opr, then send it to the dbms
 - If action is a lock opr, then check if lock can be granted
 - if so, update lock table
 - if not, delay transaction but update lock table to reflect transaction waiting
- When a transaction commits/aborts, Part I is notified and releases all locks. Part II will be notified if there are transactions waiting.
- Part II determines next transactions to be given the released locks. Those that acquired locks can be processed.

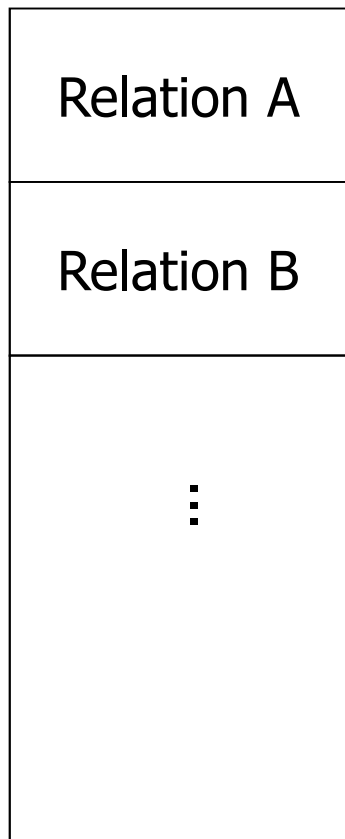
Lock table (Conceptually)



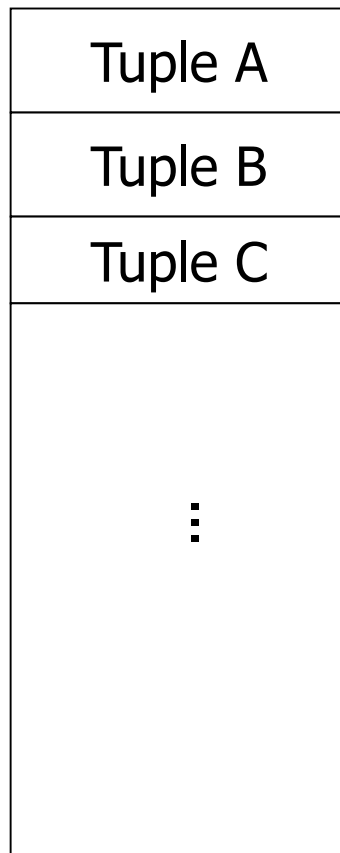
But use hash table:



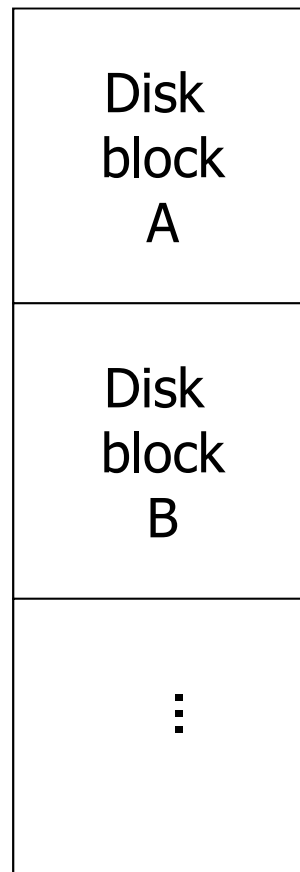
What are the objects we lock?



DB



DB



DB

Large objects (e.g.,
Relations)

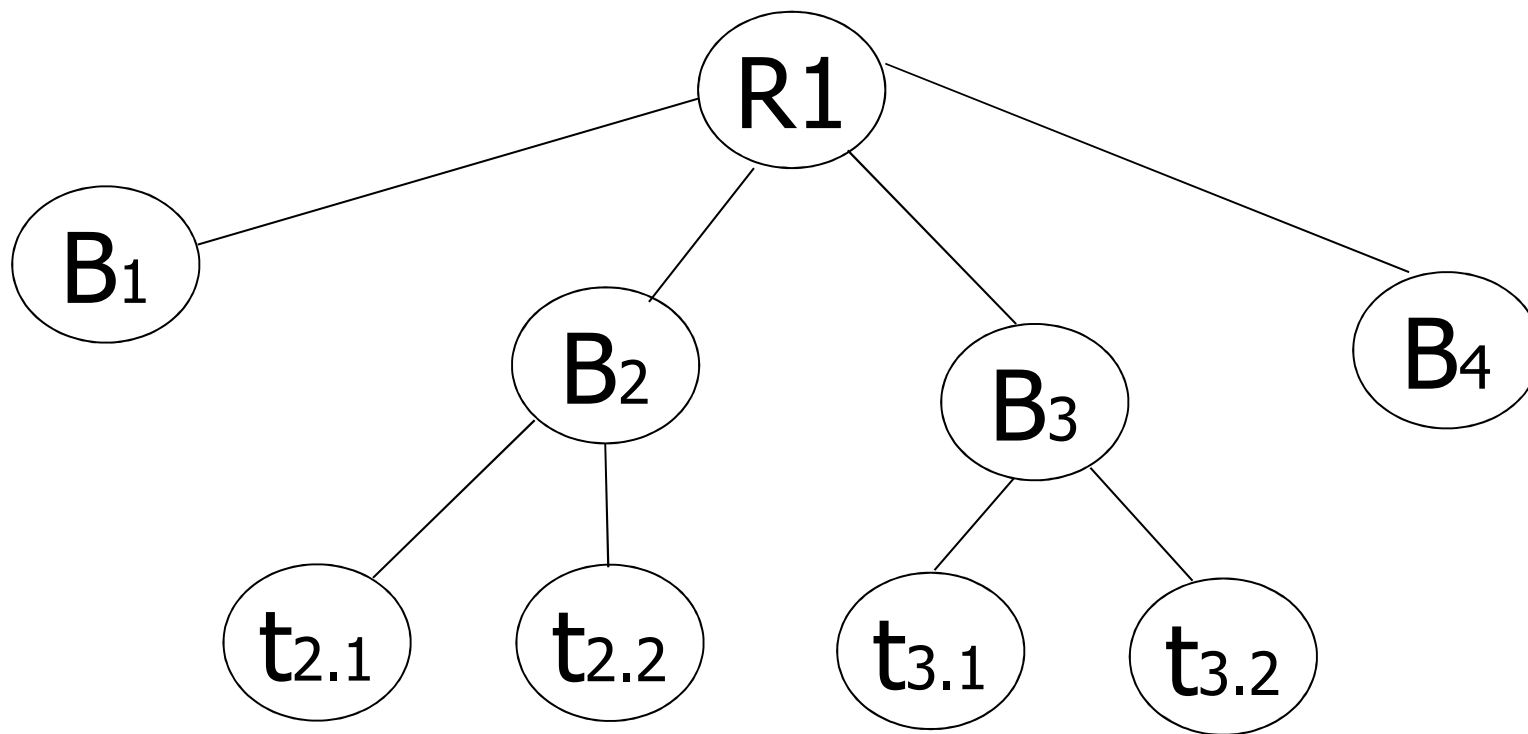
- * Need few locks
- * Low concurrency

Small objects (e.g.,
tuples, fields)

- * Need more locks
- * More concurrency

We **can** have it
both ways!!

Managing Hierarchies of Database Elements



Database
|
Tables
|
Pages
|
Tuples

Warning Protocol

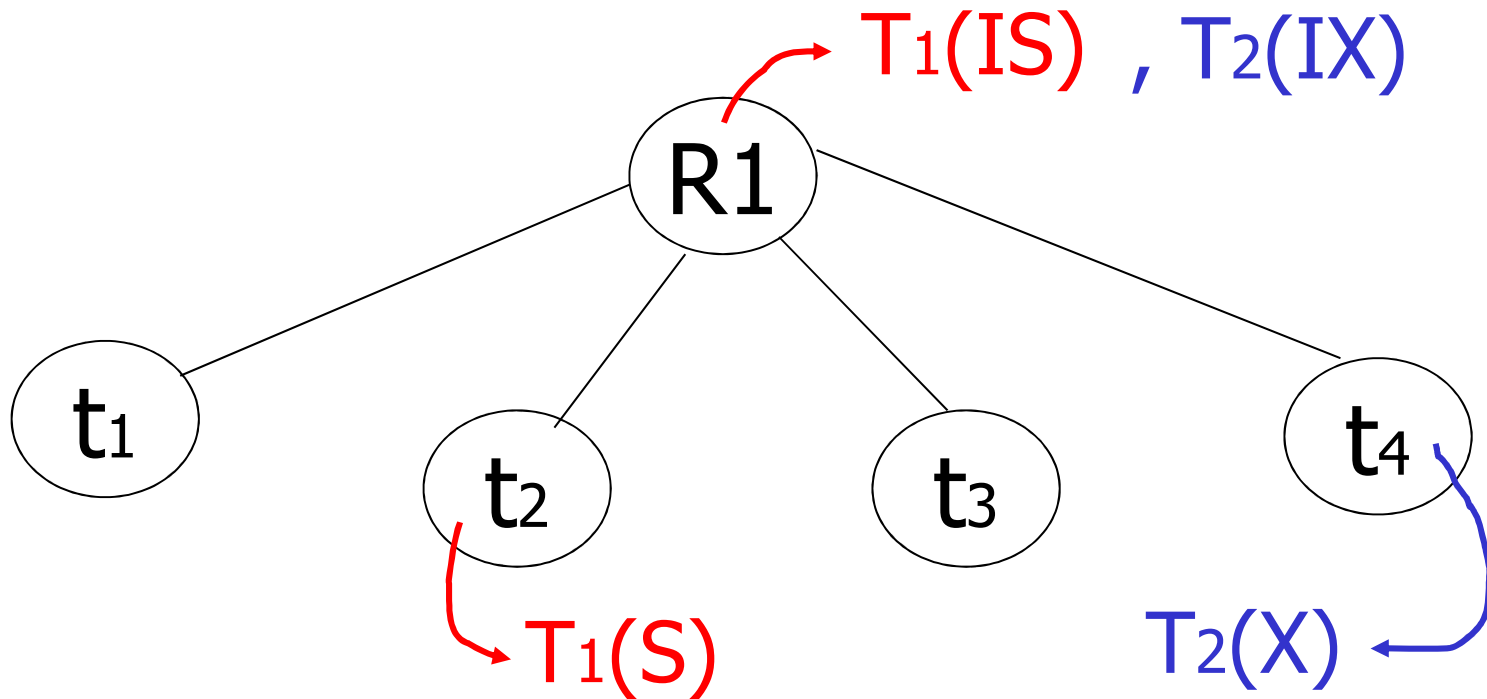
Comp

Requestor

		IS	IX	S	SIX	X
Holder	IS	T	T	T		F
	IX	T	T	F		F
	S	T	F	T		F
	SIX					
	X	F	F	F		F

- **IS** – Intent to get S lock(s) at *finer granularity*
- **IX** – Intent to get X lock(s) at *finer granularity*

Multiple Granularity: Warning Protocol



- **IS** – Intent to get S lock(s) at *finer granularity*
- **IX** – Intent to get X lock(s) at *finer granularity*

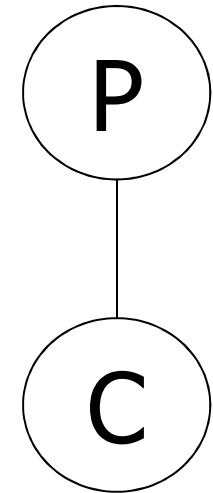
Warning Protocol

- **SIX mode:** Like S & IX at the same time

		Requestor				
		IS	IX	S	SIX	X
Holder	IS	T	T	T	T	F
	IX	T	T	F	F	F
	S	T	F	T	F	F
	SIX	T	F	F	F	F
	X	F	F	F	F	F

Does it make sense to have XIS?

Parent locked in	Child can be locked in
IS	IS, S
IX	IS, S, IX, X, SIX
S	[S, IS] not necessary
SIX	X, IX, [SIX]
X	none



Rules

- (1) Follow multiple granularity comp function
- (2) Lock root of tree first, any mode
- (3) Node Q can be locked by Ti in S or IS only if
parent(Q) locked by Ti in IX or IS
- (4) Node Q can be locked by Ti in X,SIX,IX only
if parent(Q) locked by Ti in IX,SIX
- (5) Ti is two-phase
- (6) Ti can unlock node Q only if none of Q's
children are locked by Ti

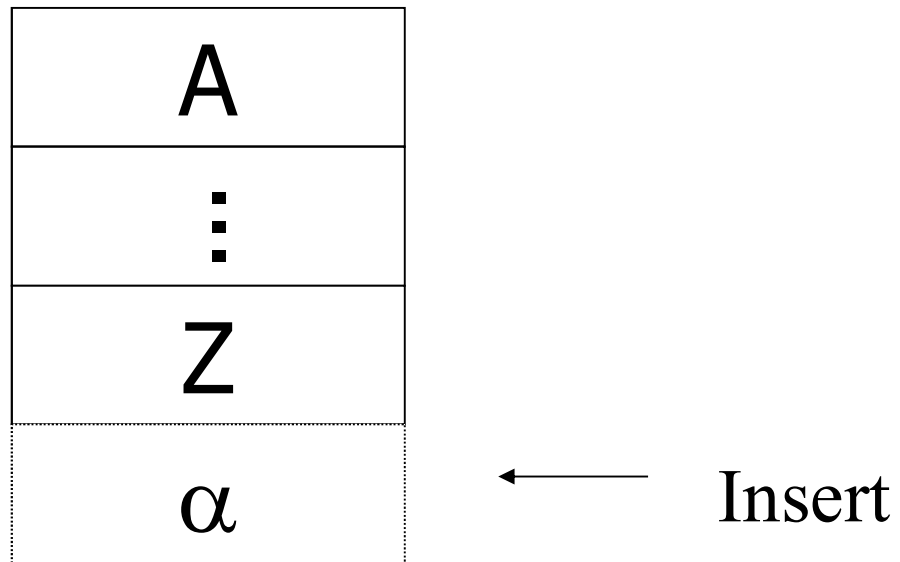
Examples – 2 level hierarchy

Tables
|
Tuples

- T1 scans R, and updates a few tuples:
 - T1 gets an SIX lock on R, then get X lock on tuples that are updated.
- T2 uses an index to read only part of R:
 - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- T3 reads all of R:
 - T3 gets an S lock on R.
 - OR, T3 could behave like T2; can use **lock escalation** to decide which.
 - Lock escalation dynamically asks for coarser-grained locks when too many low level locks acquired

	IS	IX	SIX	S	X
IS	✓	✓	✓	✓	
IX	✓	✓			
SIX	✓				
S	✓			✓	
X					

Insert + delete operations



Modifications to locking rules:

- (1) Get exclusive lock on A before deleting A
- (2) At insert A operation by T_i ,
 T_i is given exclusive lock on A

Concurrency Control Anomalies & Locking

- Dirty read problem: $W1(x)$, $R2(x)$
- Unrepeatable read problem: $R1(x)$, $W2(x)$, $R1(x)$
- Lost update problem: $R1(x)$, $R2(x)$, $W1(x)$, $W2(x)$
- But still has an anomaly: Phantom problem
 - A transaction *re-executes* a query returning a set of rows that satisfy a search condition and finds that the *set of rows* satisfying the condition has changed due to another recently committed transaction

Phantom Read Problem

Accounts

account	name	balance
100	Alice	5000
200	Bob	800
300	Carol	1000



Accounts

account	name	balance
100	Alice	5000
200	Bob	800
300	Carol	1000
400	Dave	3000

```
begin transaction;
select  name
from    Accounts
where   balance > 1000;
```

```
select  name
from    Accounts
where   balance > 1000;
commit;
```

```
begin transaction;
insert into Accounts
      values (400,'Dave',3000);
commit;
```

$R_1(100, \text{Alice}, 5000)$

$R_1(200, \text{Bob}, 800)$

$R_1(300, \text{Carol}, 1000)$

Output: {Alice}

$W_2(400, \text{Dave}, 3000)$

Commit_2

$R_1(100, \text{Alice}, 5000)$

$R_1(200, \text{Bob}, 800)$

$R_1(300, \text{Carol}, 1000)$

$R_1(400, \text{Dave}, 3000)$

Commit_1

Output: {Alice, Dave}

Phantom Update Problem

Example: relation R (E#,name,...)

constraint: E# is key

use tuple locking

R	E#	Name
o1	55	Smith	
o2	75	Jones	

T₁: Insert <99,Gore,...> into R

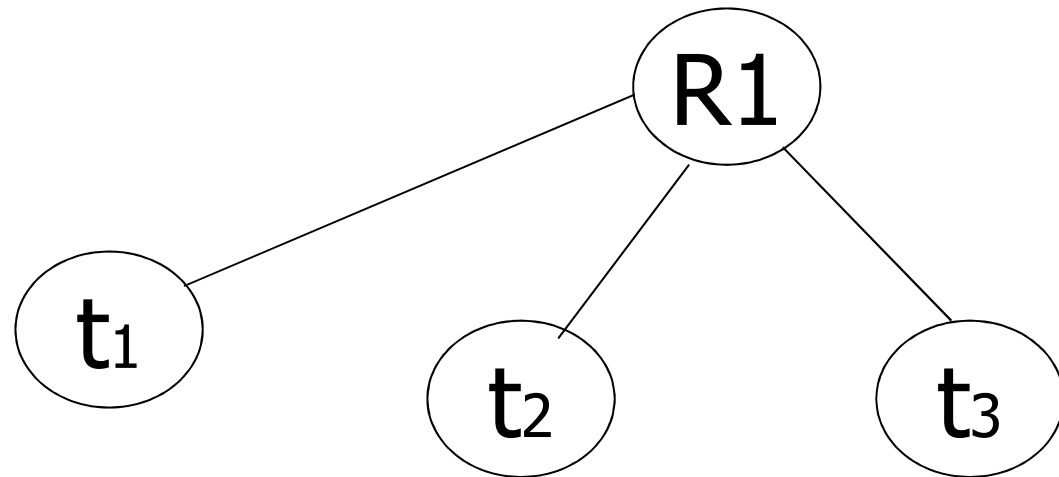
T₂: Insert <99,Bush,...> into R

T ₁	T ₂
S1(o1)	S2(o1)
S1(o2)	S2(o2)
Check Constraint	Check Constraint
-- no such key, ok to insert	-- no such key, ok to proceed
⋮	⋮
Insert o3[99,Gore,...]	Insert o4[99,Bush,..]

Violation of key constraint!

Solution

- Use multiple granularity tree
- Before insert of node Q,
lock parent(Q) in
X mode



Back to example

T₁: Insert<99,Gore>

T₁

X₁(R)

Check constraint

Insert<99,Gore>

U(R)

T₂: Insert<99,Bush>

T₂

X₂(R)

delayed

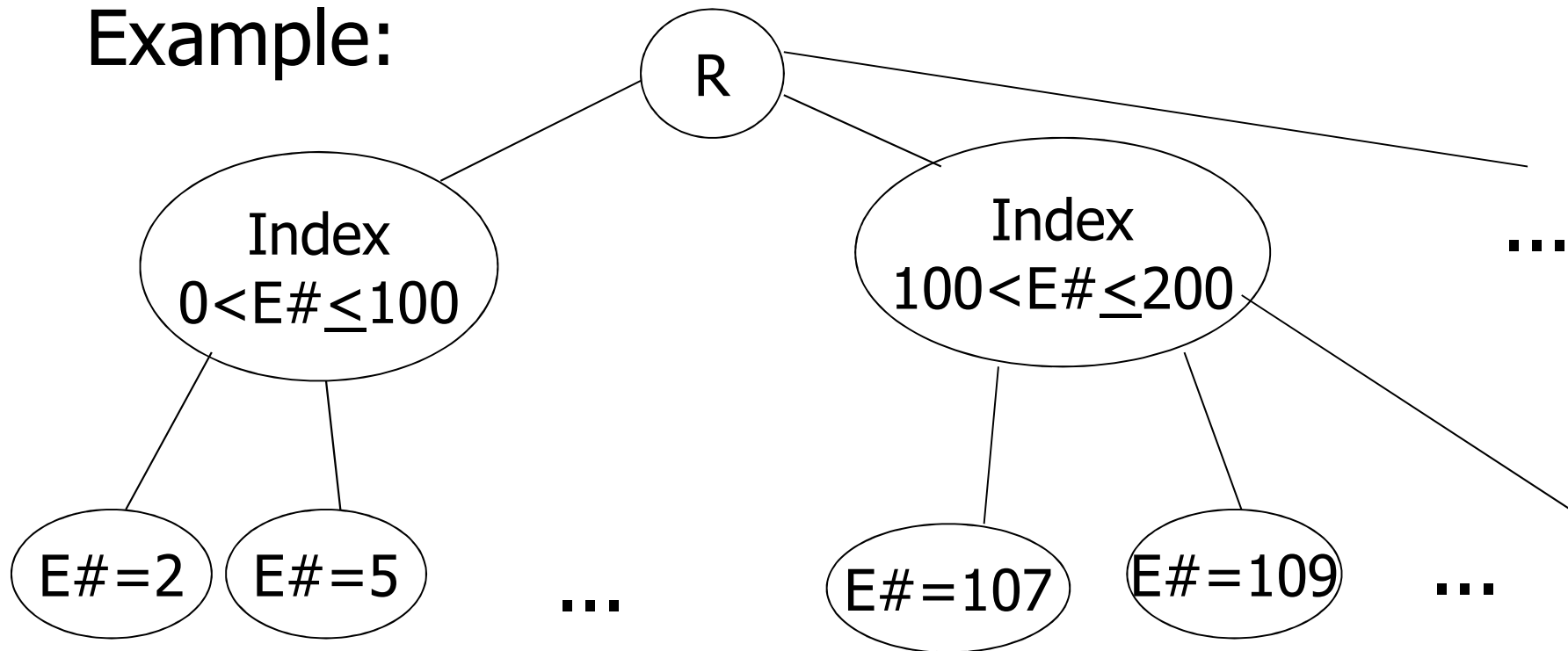
X₂(R)

Check constraint

Oops! e# = 99 already in R!

Instead of using R, can use index on R:

Example:



ANSI SQL Isolation Levels

Isolation Level	Dirty Read	Unrepeatable 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

- SQL's SET TRANSACTION ISOLATION LEVEL command

```
BEGIN TRANSACTION;
```

```
SET TRANSACTION ISOLATION LEVEL
```

```
{ READ UNCOMMITTED |  
  READ COMMITTED |  
  REPEATABLE READ |  
  SERIALIZABLE };
```

```
.....
```

```
COMMIT;
```

- In many DBMSs, the default isolation level is READ COMMITTED
- Isolation level: per transaction and “eye of the beholder”

Welcome to the real world!

Default and maximum isolation levels for ACID and NewSQL databases as of Jan 2013 (provided by Joe Hellerstein)

Database	Default	Maximum
Action Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	RR
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	S
MySQL 5.6	RR	S
MemSQL 1b	RC	RC
MS SQL Server 2012	RC	S
NuoDB	CR	CR
Oracle 11g	RC	SI
Oracle Berkeley DB	S	S
Oracle Berkeley DB JE	RR	S
Postgres 9.2.2	RC	S
SAP HANA	RC	SI
ScaleDB 1.02	RC	RC
VoltDB	S	S
RC: read committed, RR: repeatable read, SI: snapshot isolation, S: serializability, CS: cursor stability, CR: consistent read		

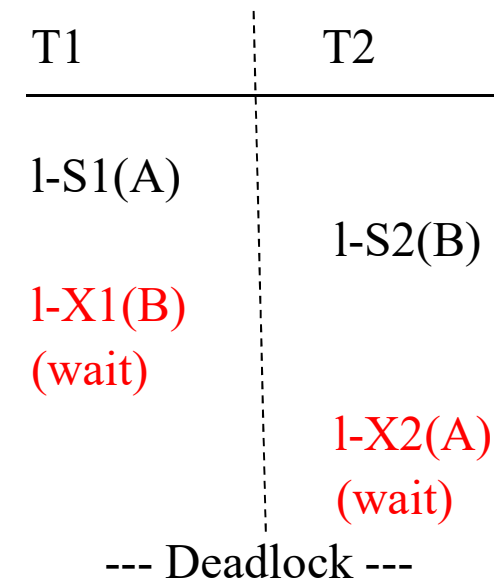
Deadlocks

Deadlocks

- Deadlocks: cycle of Xacts waiting for locks to be released by each other

- Example:
 - T1 requests S-lock on A and is granted
 - T2 requests S-lock on B and is granted
 - T1 requests X-lock on B and is blocked
 - T2 requests X-lock on A and is blocked

- Solutions:
 - Simple timeout mechanism
 - Detection
 - Wait-for graph
 - Prevention
 - Wait-die
 - Wound-wait

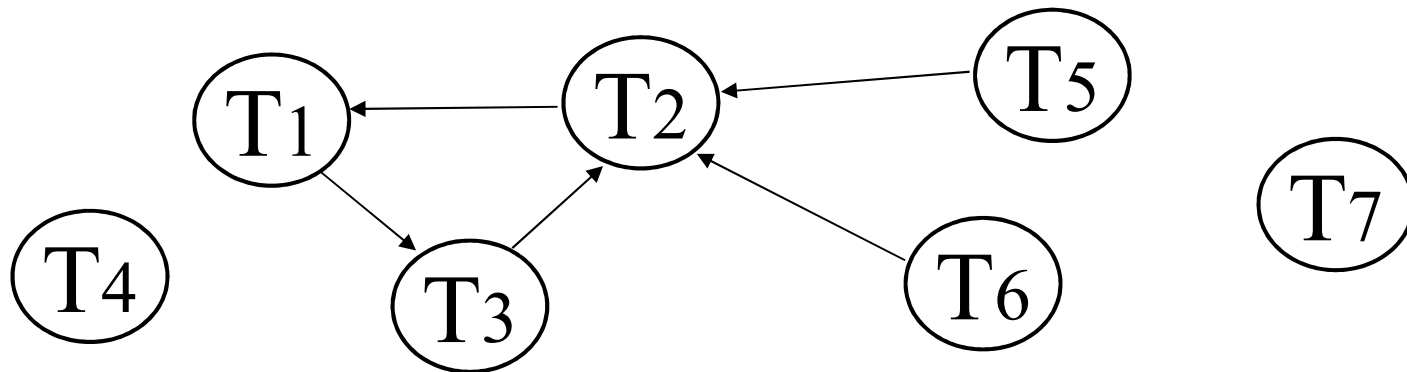


Deadlock Detection

- Build Wait-For graph
 - Nodes represent active Xacts
 - Add an edge $T_i \rightarrow T_j$ if T_i is waiting for T_j to release lock
- Use lock table structures
- Build incrementally or periodically
 - Adds an edge when it queues a lock request
 - Updates edges when it grants a lock request
- When a *cycle* is found, there is a *deadlock*

Deadlock Detection

- Breaks a deadlock by aborting a Xact in cycle
 - How to determine the victim?
 - Random
 - Most “connected”
 - Workload/time-based
 - ...



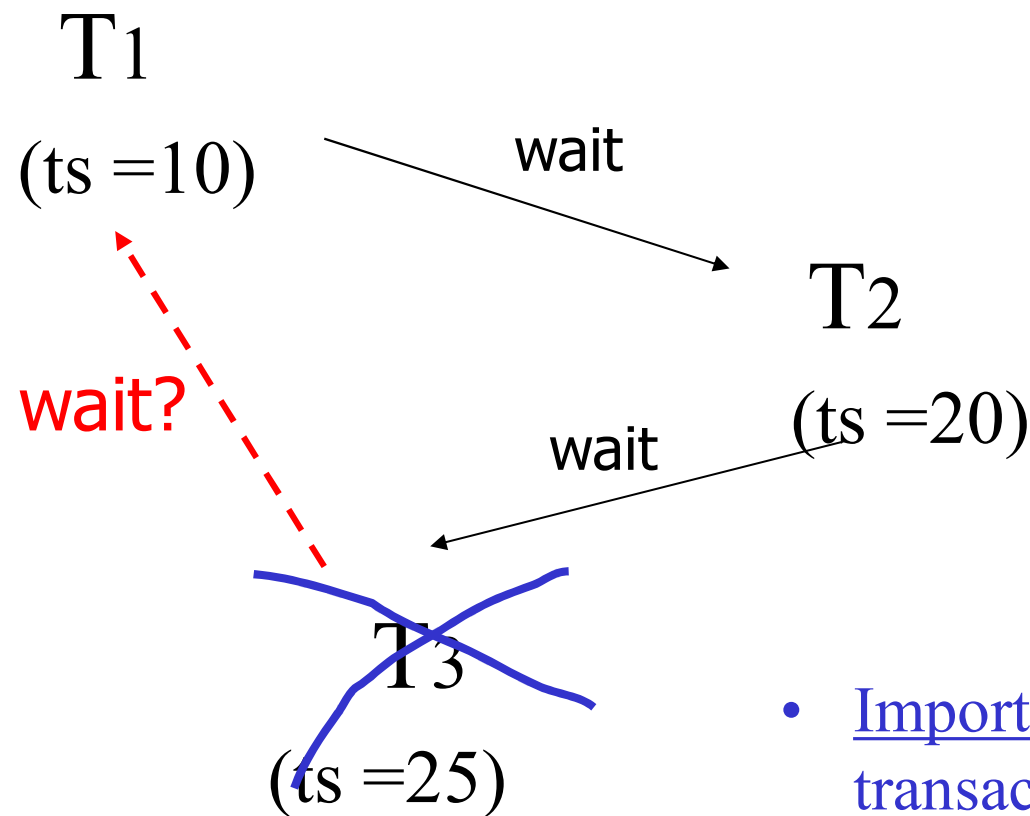
Deadlock Prevention

- Transactions given a timestamp when they arrive $ts(T_i)$
 - An older Xact has a smaller timestamp
- Suppose T_i requests for a lock that conflicts with a lock held by T_j
- Two possible deadlock prevention policies:
 - Wait-die policy: lower-priority Xacts never wait for higher-priority Xacts
 - Wound-wait policy: higher priority Xacts never wait for lower-priority Xacts

Prevention Policy	T_i has higher priority	T_i has lower priority
Wait-die	T_i waits for T_j	T_i aborts
Wound-wait	T_j aborts	T_i waits for T_j

Wait-die Example:

- T_i can only wait for T_j if $ts(T_i) < ts(T_j)$
...else die (i.e., abort)



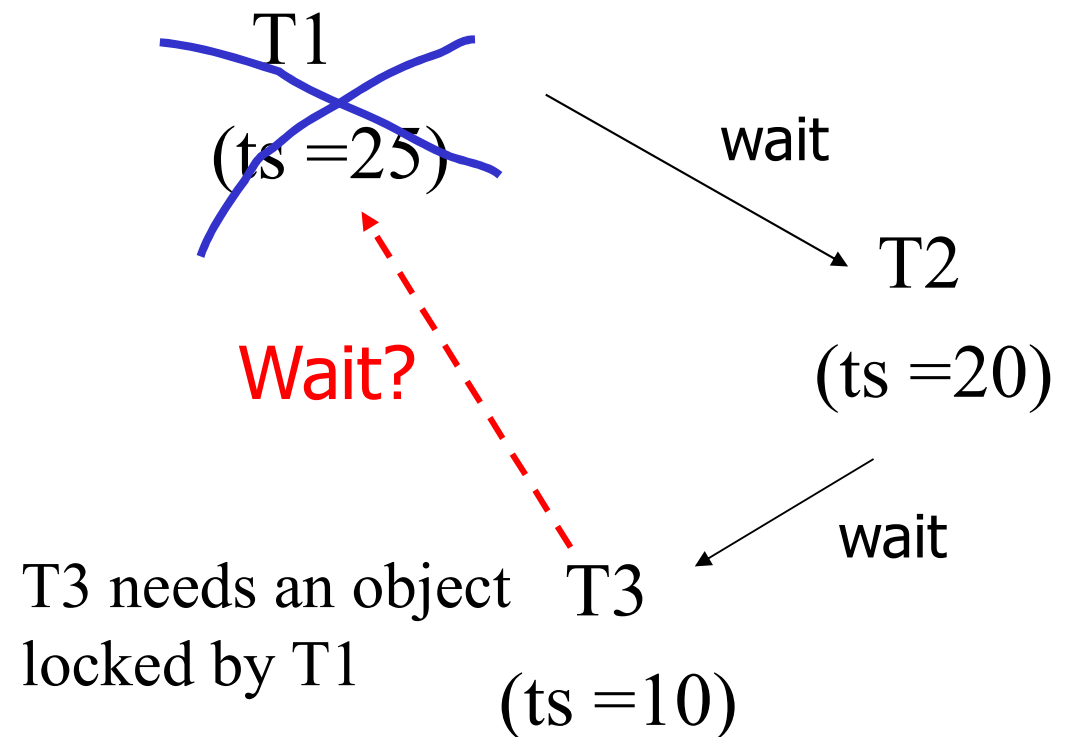
- Important detail: If a transaction re-starts, make sure it gets its original timestamp. **Why?**

Deadlock Prevention: Wound-wait

- T_i wounds T_j if $ts(T_i) < ts(T_j)$
else T_i waits

“Wound”: T_j rolls back and gives lock to T_i

Wound-wait Example



Optimistic Concurrency Control

Optimistic & Lock Free Concurrency Control: Validation-based Protocol

Transactions have 3 phases:

(1) Read

- All DB values read/written
 - Make a copy in temporary (local) storage
 - All updates/writes on local storage (no in-place updates/writes)
- No locking
- Maintains *read-set* (RS) and *write-set* (WS)

(2) Validate

- Check if schedule so far is serializable (ensure no conflict between RS/WS sets of transactions)

(3) Write

- If validate ok, write to DB values of WS; if not, “roll-back”

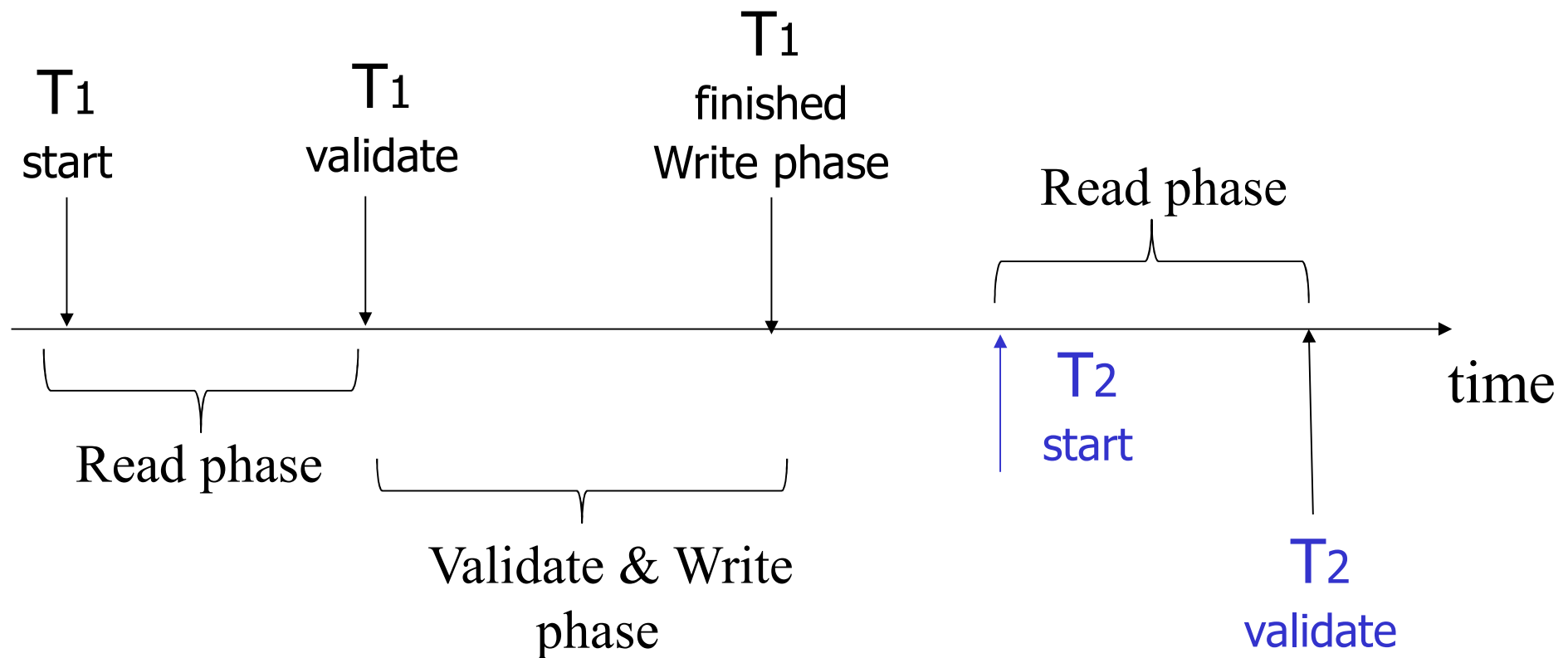
Validation-based Protocol

- Timestamps
 - $\text{start}(T)$: start of read phase of T
 - $\text{validate}(T)$: start of validate phase of T
 - $\text{finish}(T)$: end of write phase
- Timestamp of **transaction**, $\text{ts}(T)$: **$\text{validate}(T)$**
- Transactions are serialized using $\text{ts}(T)$
 - If $\text{ts}(T1) < \text{ts}(T2) < \text{ts}(T3) \dots$, then $T1, T2, T3, \dots$ is the validation order, i.e., the resulting schedule will be conflict equivalent to serial order $T1, T2, T3, \dots$
- Cascadeless schedule
- No locking! – no deadlocks!
- Starvation (rolled-back transactions restart with a new timestamp!)

Validation tests

- Validating T_i
 - $\forall T_j$ such that $ts(T_j) < ts(T_i)$ // Compare with transactions that have validated
 - Rule 1: $Finish(T_j) < start(T_i)$ **OR**
 - Rule 2: If $Start(T_i) < Finish(T_j) < validation(T_i)$, then $read-set(T_i) \cap write-set(T_j) = \emptyset$ **OR**
 - Rule 3: If T_j finishes after T_i starts and validates, then $read-set(T_i) \cap write-set(T_j) = \emptyset$ **AND** $write-set(T_i) \cap write-set(T_j) = \emptyset$
- Validation passes if the above is true
 - First rule says they are serial
 - Second and third say there are no conflicts

Example of what validation must **allow (Rule 1)**:



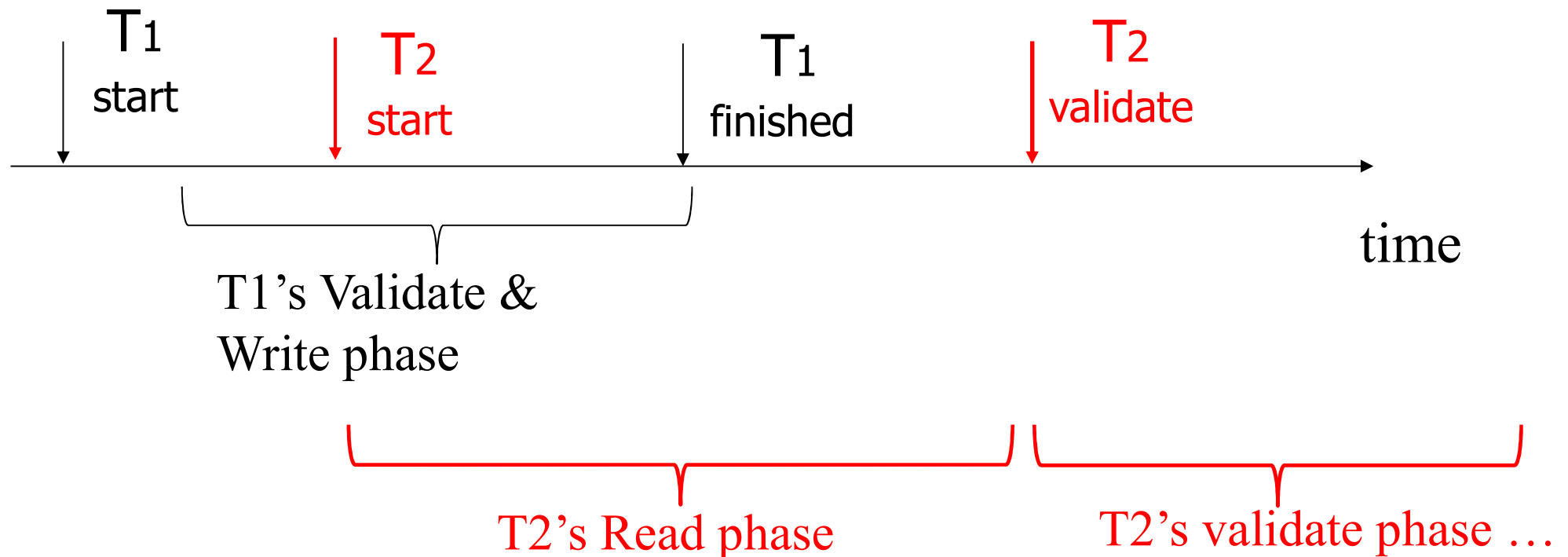
Example of what validation must prevent:

$RS(T_1) = \{B\}$

$WS(T_1) = \{B, D\}$

$RS(T_2) = \{A, B\} \neq \phi$

$WS(T_2) = \{C\}$



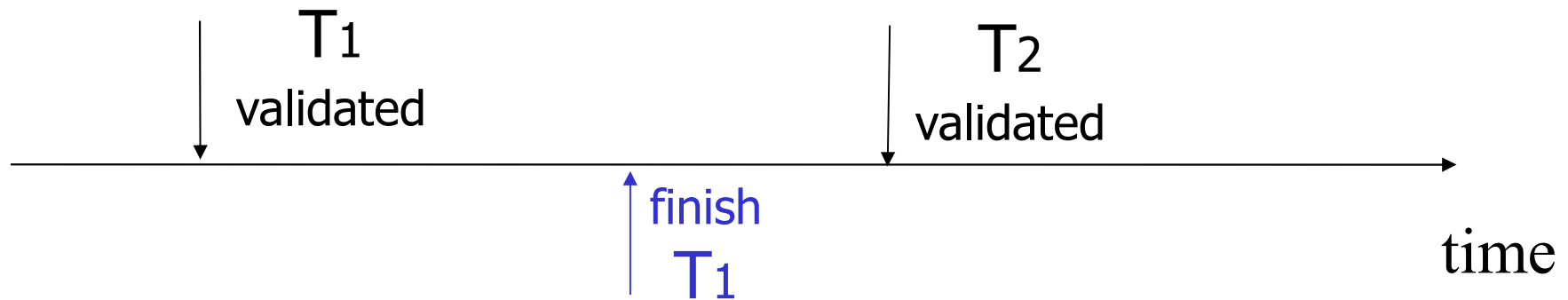
Another thing validation must **allow (Rule 2):**

$RS(T_1) = \{A\}$

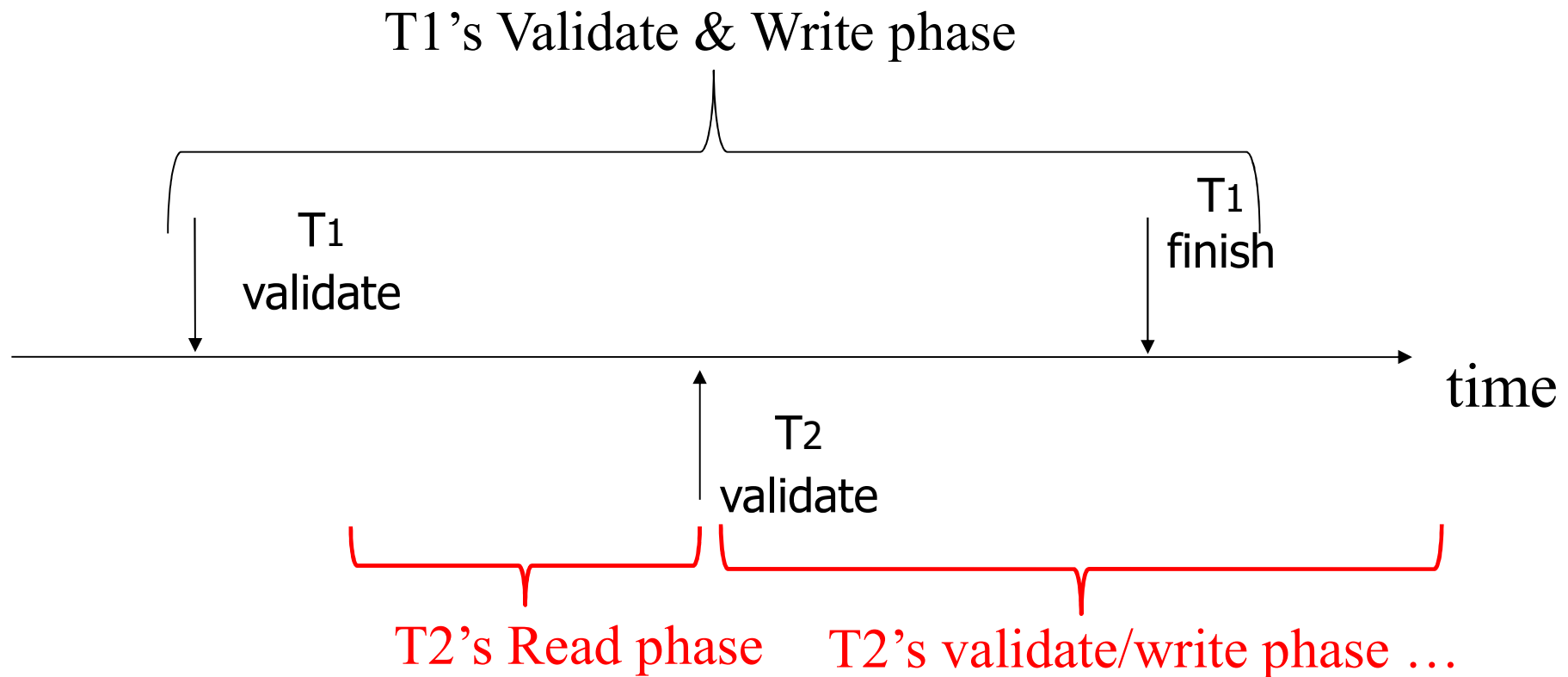
$RS(T_2) = \{A, B\}$

$WS(T_1) = \{D, E\}$

$WS(T_2) = \{C, D\}$



Another thing validation must prevent (?):



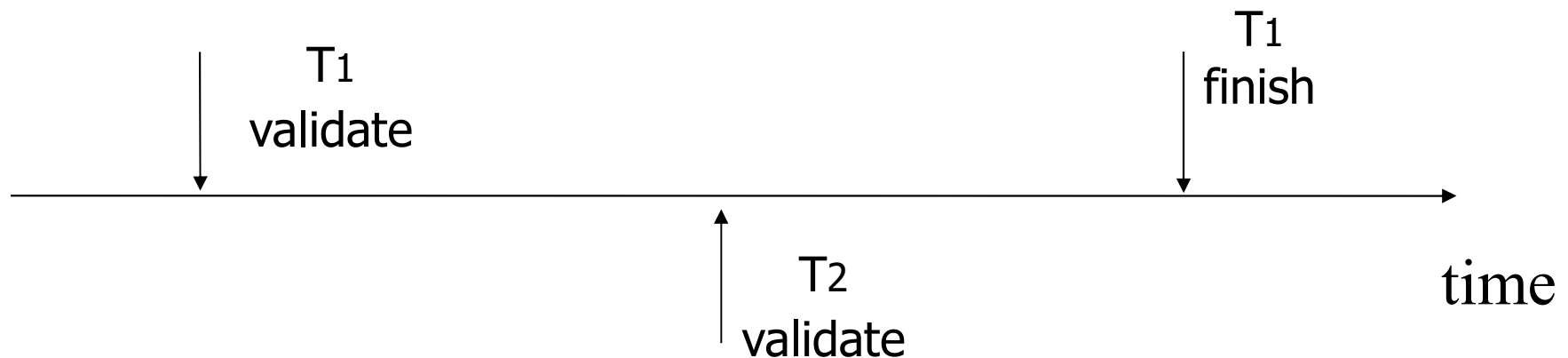
Another thing validation must prevent:

$RS(T_1) = \{A\}$

$RS(T_2) = \{A, B\}$

$WS(T_1) = \{D, E\}$

$WS(T_2) = \{C, D\}$



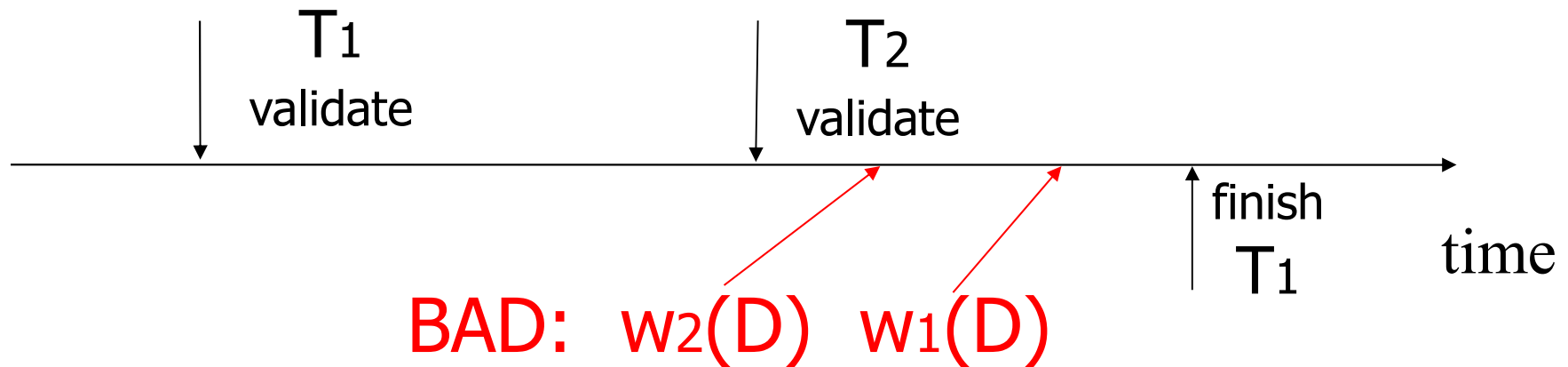
Another thing validation must prevent:

$RS(T_1) = \{A\}$

$RS(T_2) = \{A, B\}$

$WS(T_1) = \{D, E\}$

$WS(T_2) = \{C, D\}$



To implement validation, system keeps two sets:

- FIN = transactions that have finished phase 3 (and are all done)
- VAL = transactions that have successfully finished phase 2 (validation)

Validation rules for T_j :

(1) When T_j starts phase 1:

$\text{ignore}(T_j) \leftarrow \text{FIN}$

(2) At T_j Validation:

if $\text{Valid}(T_j)$ then

$[\text{VAL} \leftarrow \text{VAL} \cup \{T_j\};$

do write phase;

$\text{FIN} \leftarrow \text{FIN} \cup \{T_j\}]$

Valid(T_j):

For $T_i \in \text{VAL} - \text{IGNORE}(T_j)$ DO

IF [$\text{WS}(T_i) \cap \text{RS}(T_j) \neq \emptyset$ OR “if they wrote
and we read”

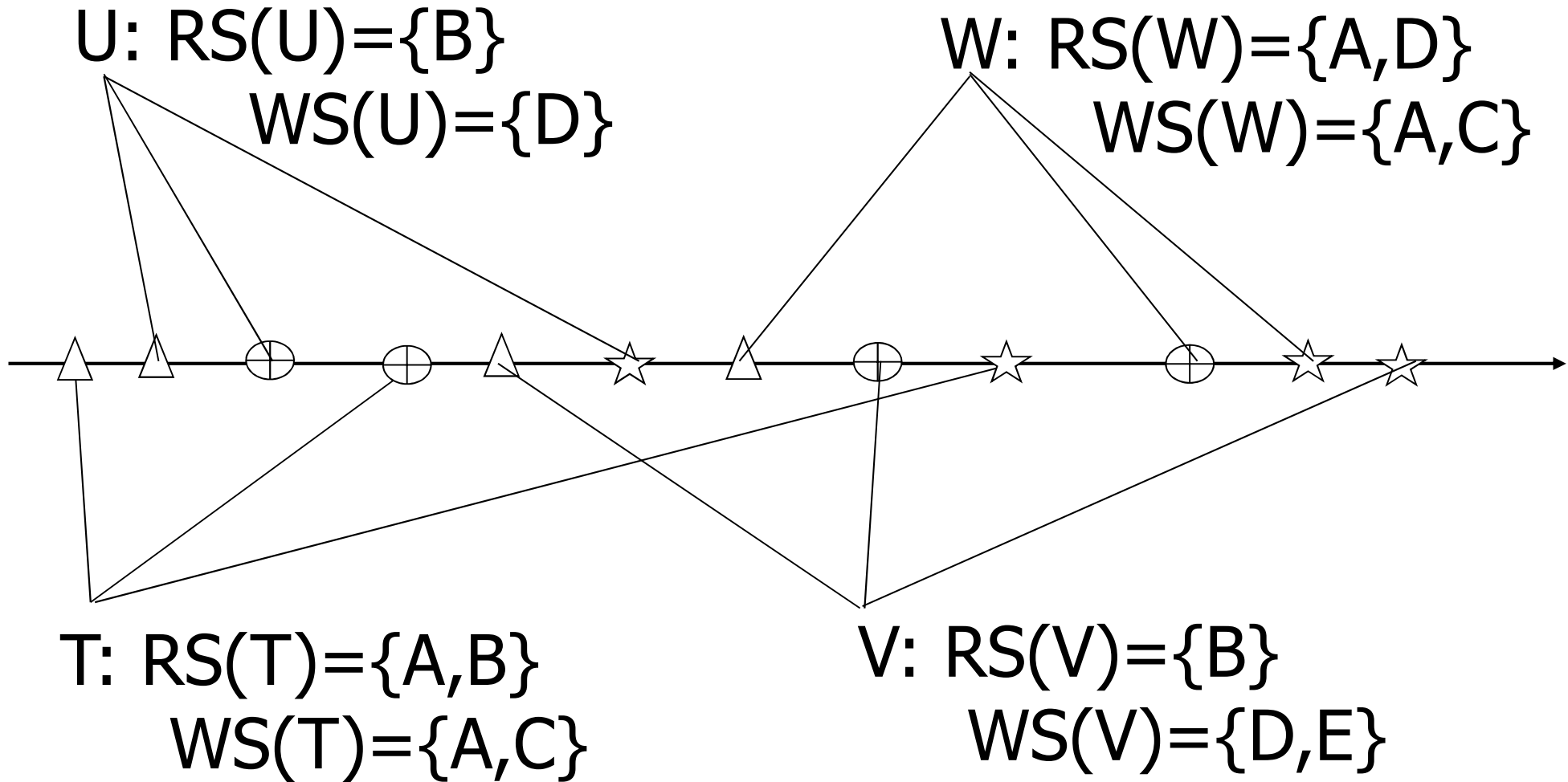
($T_i \notin \text{FIN}$ AND $\text{WS}(T_i) \cap \text{WS}(T_j) \neq \emptyset$)]

THEN RETURN false; T_i is not done and
we both wrote to

RETURN true; the same items”

Exercise:

△ start
⊕ validate
☆ finish



Summary

- Have studied lock-based CC mechanisms
 - 2 PL
 - Multiple granularity
 - Deadlock
 - Validation-based schemes
- There are also non-locking based CC (timestamp) schemes, e.g., Multiversion (Snapshot Isolation)