CHAPTER 2

TRANSACTION – CONCURRENCY CONTROL TECHNIQUES

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Goals

- Could operate on a database consistently using transactions.
- Comprehend and apply concurrency control techniques provided by DBMSs.

Outline

1. Introduction
2. Transaction
3. Concepts about concurrency control: schedules, serial schedules, serializable schedules, …
4. Isolation levels
5. Deadlock
6. How to control concurrent transactions using locking methods provided by a DBMS?
7. Concurrency Control Based on Timestamp Ordering
8. Multiversion Concurrency Control Techniques
9. Validation Concurrency Control Techniques

2. Transaction

- Definition
- Properties
- How to design a transaction

Transaction

A transaction is an executing program that forms a logical unit of database processing.

A transaction includes one or more database access operations: insertion, deletion, modification, or retrieval operations.

Properties of Transaction: ACID

- Atomicity

  A transaction is an atomic unit of processing, it is either performed in its entirety or not performed at all.
- Consistency

  A transaction is consistent if its complete execution transforms the database from one consistent state to another.
- Isolation

  A transaction should appear as though it is being executed in isolation from other transactions.
- Durability

  The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

Transaction

- Given two relations:
  - LOP (<u>MALOP</u>, TENLOP, SISO)
  - SV (<u>MASV</u>, TENSV, MALOP)
- Intergrity constraint: The number of students (SISO) must reflect the students belonging to that class.
- How to insert a new student into a class:

```
 Insert_Student (v_masv, v_tensv, v_malop)
Begin Tran
    If not exists v_malop
        Rollback tran
    Insert into SV (v_masv,v_tensv,v_malop)
    Update    LOP    Set     SISI= SISO + 1
    WhereMALOP    =    v_malop
Commit Tran
```

T-SQL for Transaction

BEGIN TRANSACTION	**Marks the beginning of transaction execution.**
COMMIT TRANSACTION	**Marks a successful end of the transaction.**
ROLLBACK TRANSACTION	**Signals that the transaction has ended unsuccessfully.**

Remarks

- Test if no proper rights, integrity constraint violation,or deadlock, …
- Global variable @@ERROR

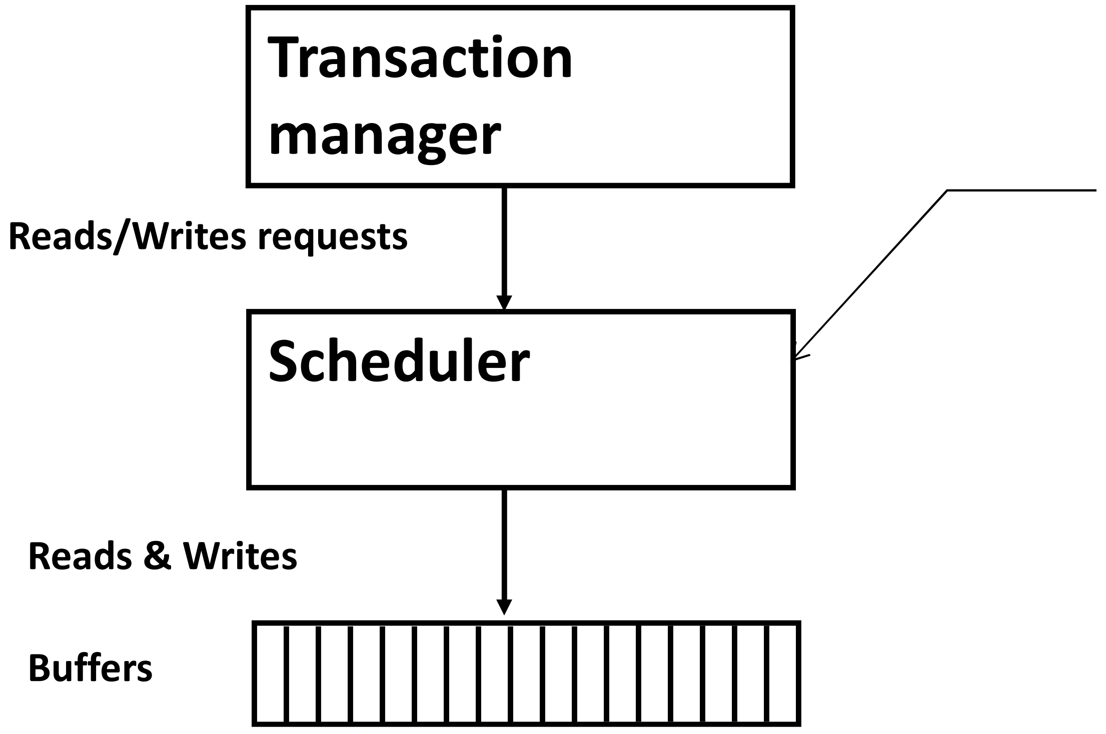
  =0 : there is no fault, ≠ 0 : there is some fault
- A transaction cannot roll back by itself, the transaction designer must control the cases in which it must be rolled back.
- Test the value of @@ERROR after each update operations, then process the error(s).
- Global variant @@ROWCOUNT
  - Test the value of @@ROWCOUNT to know if any row affected by the previous operation.

3. CONCURRENCY CONTROL

- Introduction
- Problems caused by concurrently executing transactions

Concurrency control

- Interactions among transactions can cause the database state to become inconsistent, even when the transactions individually preserver correctness of the state, and there is no system failure.



Scheduler receives reads and/ or writes from transactions. Scheduler decides:

- Execute them.
- Delay them, because it's not safe for them to be executed immediately.
- Or even abort the transaction that issues the request (in some concurrency control techniques)

Problems with LOST UPDATE

A lost update occurs when two different transactions are trying to update the same column on the same row within a database at the same time. The result of the first transaction is then "lost", as it is simply overwritten by the second transaction.

T1	T2
Begin Tran	
Read A	
	Begin Tran
	Read A
A:=A+10	
Write A	
	A:=A*100
	Write A
	Commit Tran
Commit Tran	

Problems with LOST UPDATE

Ex 2:

T1	T2
Begin Tran	
	Begin Tran
Write A	
	Write A
	Commit Tran
Commit Tran	

Problems with UNCOMMITED DATA/ DIRTY READ

A Dirty read is the situation when a transaction reads a data that has not yet been committed.

T1	T2
Begin Tran	
Read A	
A:=A+10	
Write A	
	Begin Tran
	Read A
	Print A
	Commit Tran
Rollback Tran	

Problems with UNREPEATABLE DATA

Non Repeatable read occurs when a transaction reads the same row twice, and get a different value each time.

T1	T2
Begin Tran	
Read A	
	Begin Tran
	Read A
	A:=A+10
	Write A
	Commit Tran
Read A	
Commit Tran	

Problems with PHANTOM

Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different.
For example, suppose transaction T1 retrieves a set of rows that satisfy some search criteria. Now, Transaction T2 generates some new rows that match the search criteria for transaction T1. If transaction T1 re-executes the statement that reads the rows, it gets a different set of rows this time.

T1	T2
Begin Tran	
Select * From SV	
	Begin Tran
	Insert into SV values (...)
	Commit Tran
Select * From SV	
Commit Tran	

4. SCHEDULES

- Concepts:
  - Schedules
  - Serial schedules
  - Serializable schedules
  - Recoverable schedules
  - Schedules that avoid cascading rollback
  - Schedulers
  - Protocols

Schedule & Correctness principle

1. Schedule

   A schedule is a time-ordered sequence of actions taken by one or more transactions.

2. Correctness principle:
   - Every transaction, if executed in isolation (without any other transactions running concurrently), will transform any consistent state to another consistent state.
   - The schedule of actions of concurrent transactions NEED to produce the same result as if the transactions executed one-at-a-time.

T1	T2
Read (A,t)	Read (A,s)
t:=t+100	s:=s*1
Write(A,t)	Write(A,s)
Read(B,t)	Read(B,s)
t:=t+100	s:=s*1
Write(B,t)	Write(B,s)

Definition of T1, T2

T1	T2
Read (A,t)	
t:=t+100	
Write(A,t)	
	Read(A,s)
	s:=s*1
	Write(A,s)
	Read (B,s)
	s:=s*1
	Write(B,s)
Read(B,t)	
t:=t+100	
Write(B,t)	

Schedule 1

T1	T2
Read (A,t)	
t:=t+100	
Write(A,t)	
	Read(A,s)
	s:=s*1
	Write(A,s)
Read(B,t)	
t:=t+100	
Write(B,t)	
	Read (B,s)
	s:=s*1
	Write(B,s)

Schedule 2

Serial / Serializable schedules

- A schedule is serial if its actions consist of all the actions of one transaction, then all the actions of another transaction, and so on.
- A serial schedule transforms the database from any consistent state to another consistent state.
  - We say, every serial schedule preserve consistency of the database state.
- A schedule is serializable if its effect on the database state is the same as that of some serial schedule, regardless of what the initial state of the database is.

Example of serial schedule

T1	T2
Read (A,t)	
t:=t+100	
Write(A,t)	
Read(B,t)	
t:=t+100	
Write(B,t)	
	Read(A,s)
	s:=s*1
	Write(A,s)
	Read (B,s)
	s:=s*1
	Write(B,s)
	Read(A,s)

Examples of serializable schedules

T1	T2	A	B
T1<T2: A=B=250		25	25
T2< T1: A=B=150			
Read (A,t)			
t:=t+100			
Write(A,t)		125	
	Read(A,s)		
	s:=s*2		
	Write(A,s)	250	
	Read (B,s)		
	s:=s*2		
	Write(B,s)		50
Read(B,t)			
t:=t+100			
Write(B,t)			150

This schedule is not serializable

T1	T2	A	B
T1<T2: A=B=125		25	25
T2< T1: A=B=125			
Read (A,t)			
t:=t+100			
Write(A,t)		125	
	Read(A,s)		
	s:=s*1		
	Write(A,s)	125	
	Read (B,s)		
	s:=s*1		
	Write(B,s)		25
Read(B,t)			
t:=t+100			
Write(B,t)			125

The schedule is serializable but it is not a serial schedule

Serializable schedule

- We can consider in detail the operations performed by the transactions, to determine whether or not a schedule is serializable.
- However, the scheduler does not consider in detail the computation undertaken by transactions to determine whether or not a schedule is serializable.

Notation

- r_i (X) : transaction T_i reads the database element X.
- w_i (X) : transaction T_i writes the database element X.
- A transaction T_i is a sequence of actions with subcript i.
- A schedule S of a set of transactions is a sequence of actions, in which for each transaction T_i, the actions of T_i appear in S in the same order that they appear in the definition of T_i itself.

Recoverable schedule

T1	T2
Begin Tran	
Read A	
Write A	
	Begin Tran
	Read A
	Commit
Read B	
Commit	

$\Rightarrow$

T1	T2
Begin Tran	
Read A	
Write A	
	Begin Tran
	Read A
Read B	
Commit	
	Commit

When T1 has to roll back, T2 has committed already. T2 cannot roll back ⇒ Unrecoverable schedule.

Recoverable Schedule

- A schedule where, for each pair of transactions Ti and Tj, if Tj reads a data item previously written by Ti, then the commit operation of Ti precedes the commit operation of Tj.

Cascading rollback schedule

T1	T2	T3
Read A		
Read B		
Write A		
	Read A	
	Write A	
		Read A

When T1 fail, T2 has to be rolled back, T3 has to be rolled back too.
A phenomenon, in which an uncommitted transaction has to be rolled back because it read an item from a transaction that failed, is call cascading rollback.

Cascadeless Schedule

- Cascading rollback is quite time-consuming, since numerous transactions can be rolled back. It is important to characterize the schedules where this phenomenon is guaranteed not to occur.
- A schedule is said to be cascadeless, or to avoid cascading rollback, if every transaction in the schedule reads only items that were written by committed transactions.
- All cascadeless schedules are recoverable.

Concepts : commute or conflict

- We say that two database operations, p1 and p2, commute if, for all possible initial database states,
  - p1 returns the same value when executed in either the sequence p1, p2 or p2, p1
  - p2 returns the same value when executed in either the sequence p1, p2 or p2, p1
  - The database state produced by both sequences is the same

T1	T2
Read (A)	
	Read(A)

T1	T2
Read (A)	
	Read(B)

T1	T2
Read (A)	
	Write(B)

Two read operations on the same item commute.

Two operations on different data items always commute.

Concepts : commute or conflict

- Operations that do not commute are said to conflict.
  - A read and a write on the same item conflict.
  - Two write operations on the same item conflict.

T1	T2
Read (A)	
	Write(A)

T1	T2
Write(A)	
	Read(A)

T1	T2
Write(A)	
	Write(A)

Schedule Equivalence

- Two schedules of the same set of operations are equivalent if and only if conflicting operations are ordered in the same way in both.
- A schedule is serializable if it is equivalent to a serial schedule in the sense that conflicting operations are ordered in the same way in both.

Schedule Equivalence (cont)

- Given a schedule, S1, by interchanging commuting consecutive operations (nonconflicting operations) in S1, until obtaining a serial schedule S2, if possible. If we can do so, S1 is a serializable schedule.
  - The effect on the database state of S1 remains the same as we perform each of the nonconflicting swaps.

Concepts

- Two schedules are *conflict-equivalent* if they can be turned one into the other by a sequence of nonconflicting swaps of adjacent actions.
- A schedule is *conflict-serializable* if it is conflict-equivalent to a serial schedule.

Concepts

- Conflict-serializability is a sufficient condition for serializability. A conflict-serializable schedule is a serializable sechedule.
- Conflict-serializability is not required for a schedule to be serivalizale, but it is the condition that the schedulers in commercial systems generally use when the need to guarantee serializability.

An example

S

T1	T2	T3	T4
	1.Read A		
		2.Read A	
	3.Write B		
		4.Write A	
5.Read B			
			6.Read B
7.Read A			
8.Write C			
			9.Write A

S’

T1	T2	T3	T4
	1.Read A		
	3.Write B		
		2.Read A	
		4.Write A	
5.Read B			
7.Read A			
8.Write C			
			6.Read B
			9.Write A

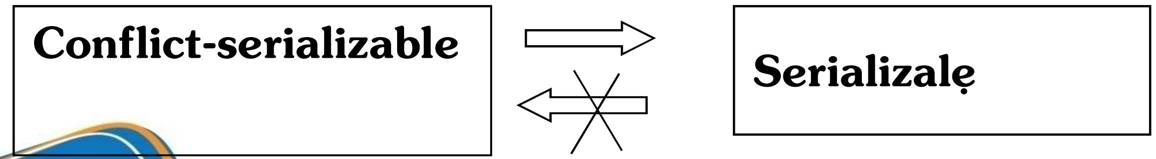
S’ is conflict - equivalent to S. S is conflict-serializable → S is serializale

Concept

- Conflict-serializability is not necessary for serializability:
  - S1:$w_1(Y);w_1(X);w_2(Y);w_2(X);w_3(X)$
  - S2:$w_1(Y);w_2(Y);w_2(X);w_1(X);w_3(X)$
- S1 leaves X with the value written by T3 and Y with the value written by T2.
- S2: the values of X written by T1 and T2 have no effect since T3 overwrites their values. The value of Y also was written by T2.
- S1 and S2 leave both X and Y with the same value.
- S1 is serial.

⇒ S2 is serializable.

⇒ However, since we cannot swap w1(Y) with w2(Y), and we cannot swap w1(X) with w2(X), therefore we cannot convert S2 to any serial schedule by swaps. That is, S2 is serializable, but not conflict-serializable.

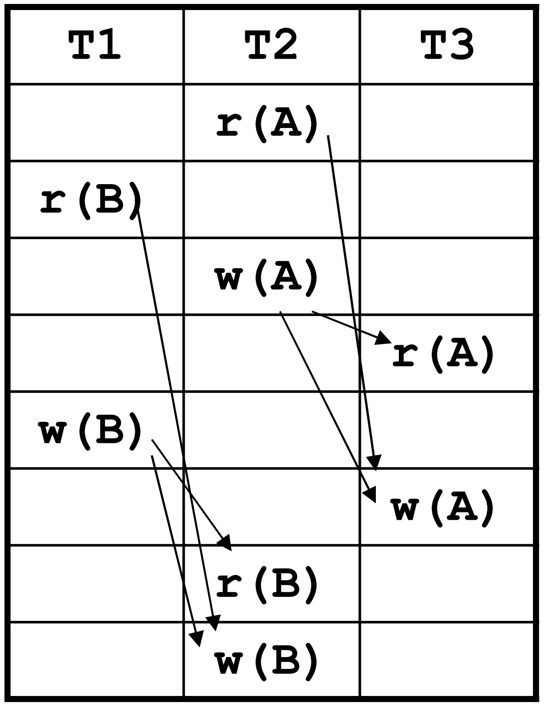


Precedence - Graph

- Precedence graph is used for testing whether a schedule S is conflict-serializable.
  - Input: Schedule S.
  - Output: S is conflict-serializable or not?
- Supose that S involves transactions Ti and Tj. Ti takes precedence over Tj, written Ti < T2, if there are actions Oin of Ti, Ojm of Tj, such that:
  - Oin is ahead Ojm in S
  - Both Oin and Ojm involve the same database element and
  - At least one of Oin and Ojm is a write operation.
- All the precedence in S can be summarized in a precedence graph.
  - The nodes of the precedence graph are the transactions of a schedule S. Each node is labeled i for the transaction Ti.
  - There is an arc from node i to node j if Ti < Tj.
- If there are any cycles in precedence graph for S, S is not conflict-serializable.
- If the precedence graph is acyclic, S is conflict-serializable. Any topological of the nodes is a conflict-equivalent serial order.

Example

- S1: $r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



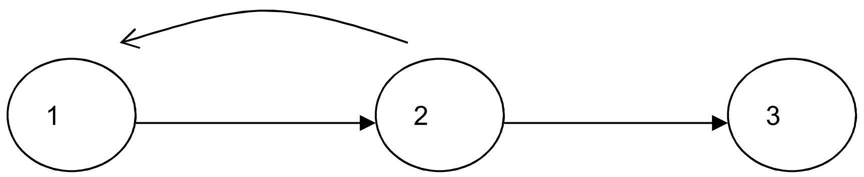
T1	T2	T3
	r(A)	
r(B)		
	w(A)	
		r(A)
w(B)		
		w(A)
	r(B)	
	w(B)	

$$\begin{tikzcd} 1 \arrow[r] & 2 \arrow[r] & 3 \end{tikzcd}$$

- The precedence graph for S1 is acyclic. So S1 is conflict-serializable, then S1 is serializable.
- The serial schedule that equivalent to S1 is T1< T2 < T3

Example

- S: $r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



T1	T2	T3
	r(A)	
r(B)		
	w(A)	
	r(B)	
		r(A)
w(B)		
		w(A)
	w(B)	

- S is not conflict-serializable.

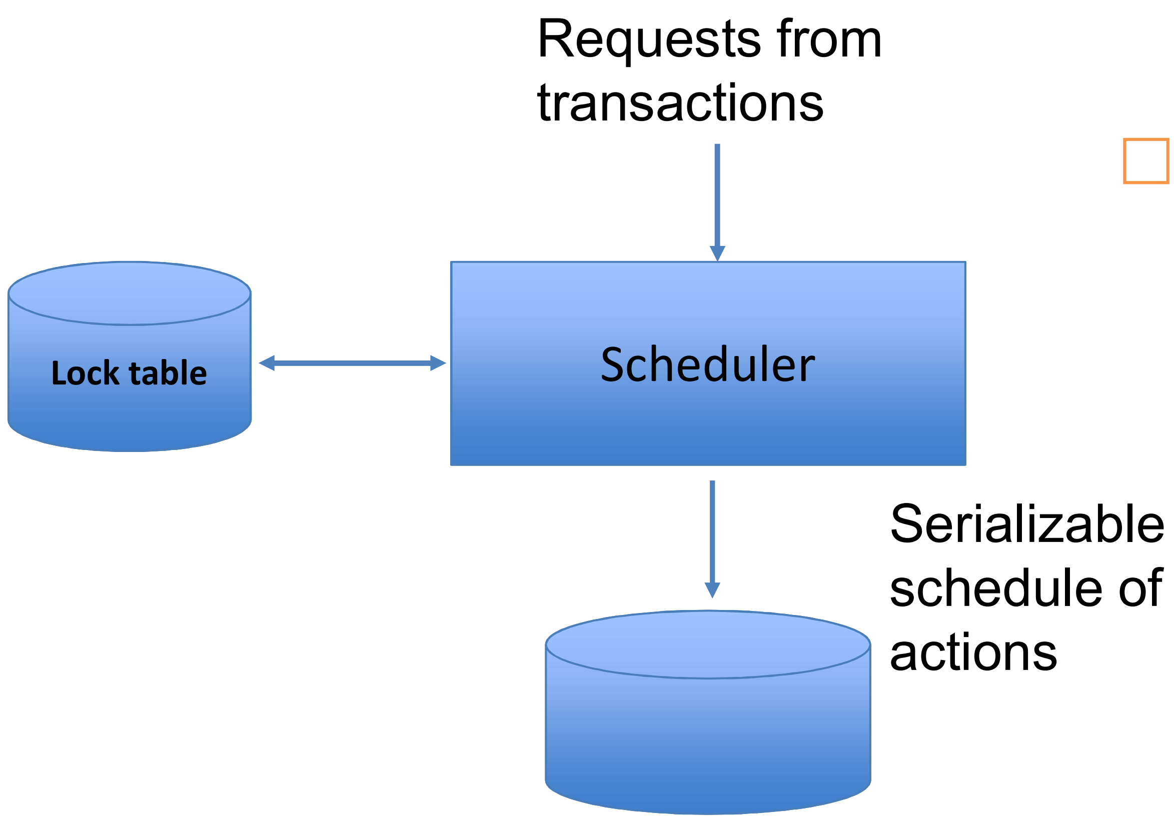
Concepts

- When a collection of transactions performing their actions in an unconstrained manner, these actions will form some schedule. It is unlikely that the schedule will be serializable.
- Scheduler is a component of a DBMS that prevent orders of actions that lead to an unserializable schedule.

ENFORCING SERIALIZABILITY BY LOCKS

1. Lock table.
2. Simple lock scheme: Lock (A), Unlock (A).
3. Two-phase locking protocol.
4. Locking systems with several lock modes: Rlock(A), Wlock(A), Unlock (A).
   - Upgrading.
   - Update lock.
   - Increment locks.
   - Lock on different granularities.
5. Tree-protocol.

Scheduler that use lock table



- The responsibility of the scheduler is to take requests from transactions and either allow them to operate on the database or defer them until such time as it is safe to allow them to execute.

Simple lock scheme

- When the scheduler uses locks, transactions must request and release locks in addition to reading and writing database elements.
- *Consistency of transactions:*
  - A transaction can only read or write an element if it previously requested a lock on that element and hasn't yet released the lock.
  - If a transaction locks an element, it must later unlock that element.
- *Legality of schedules:* no two transactions may have locked the same element without one having first release the lock.

Notation

$l_i(X)$:	**Transaction T_i requests a lock on database element X**
$u_i(X)$:	**Transaction Ti releases (unlocks) its lock on database elelement X**

- Example: T1 and T2 are consistent.
- T1:

  $l_1(A);r_1(A);A:=A+100;w_1(A);u_1(A);l_1(B);r_1(B);B:=B+10;w_1(B);u_1(B);$
- T2:

  $l_2(A);r_2(A);A:=A*2;w_2(A);u_2(A);l_2(B);r_2(B);B:=B*2;w_2(B);u_2(B);$

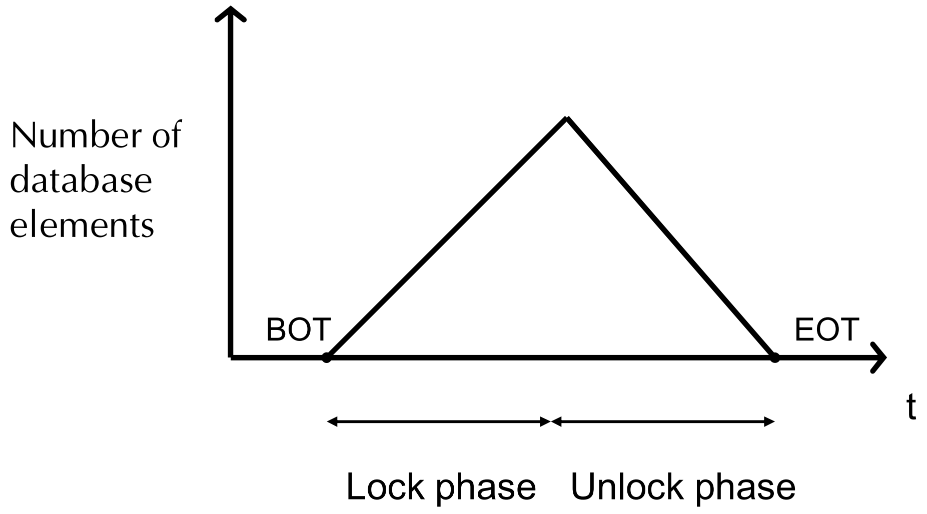
Example

T1	T2	A	B
		25	25
$l_1(A); r_1(A);$			
A:=A+100;			
$w_1(A); u_1(A);$		125	
	$l_2(A); r_2(A);$		
	A:=A*2;		
	$w_2(A); u_2(A);$	250	
	$l_2(B); r_2(B);$		
	B:=B*2;		
	$w_2(B); u_2(B);$		50
$l_1(B); r_1(B);$			
B:=B+100;			
$w_1(B); u_1(B);$			150

- **A legal schedule of consistent transactions but it is not serializable**

Two-phase locking Protocol - 2PL

- In a transaction, all lock requests predcede all unlock requests.
- 2PL is widely followed in commercial locking systems.



- In the first phase: locks are obtained; in the second phase, locks are relinquished.
- Two-phase locking is a condition on the order of actions on a transaction.
- A transaction that obeys the 2PL condition is said to be a two-phase-locked transaction, or 2PL transaction.

Example

T1
L(A)
Read(A)
L(B)
Read(B)
B:=B+A
Write(B)
UL(A)
UL(B)

T2
L(B)
Read(B)
L(A)
Read(A)
UL(B)
A:=A+B
Write(A)
UL(A)

T3
L(B)
Read(B)
B=B-50
Write(B)
UL(B)
L(A)
Read(A)
A=A+50
Write(A)
UL(A)

T4
L(A)
Read(A)
UL(A)
L(B)
Read(B)
UL(B)
Print(A+B)

T1 and T2 are two-phase-locked

T3 and T4 do not obey 2PL

- We can convert any legal schedule of consistent, two-phase-locked transactions to a conflict-equivalent serial schedule.
- n: the number of transactions in S.
  - If n = 1, there is nothing to do. S is already a serial schedule.
  - Induction: S involves n transactions: T1, T2,..., Tn. Ti is the transaction with the first unlock action in the schedule S, u_i(X).
  - It is possible to move all the read and write actions of Ti forward to the beginning of the schedule without passing any conflicting actions.
    - Consider some actions of Ti, say wi(Y). Could it be preceded in S by some conflicting action, say wj(Y). If so, the in schedule S, action uj (Y) and li(Y) must intervene, in the sequence of actions:

      S: ..., wj(Y)……, uj(Y), ..., li(Y), ..., wi(Y), ...
    - Since Ti is the first to unlock, ui(X) precedes uj(Y) in S; that is, S might look like:

      S: ..., wj(Y),..., ui(X),……, uj(Y), ..., li(Y), ..., wi(Y), ... or

      S: ..., ui(X)..., wj(Y), ..., uj(Y), ..., li(Y), ..., wi(Y), ...

    → Ti is not two-phase-locked, as we assumed.

- S can be written in the form:

(Actions of Ti) (Actions of the other n-1 transactions)

The tail of (n-1) transactions is still a legal schedule of consistent, 2PL transactions, so the inductive hypothesis applies to it. We convert the tail to a conflict-equivalent serial schedule. Now, all of S has been shown conflict-serializable.

Deadlock

- There is the potential for deadlocks in which several transactions are forced by the scheduler to wait forever for a lock held by another transaction.

Deadlock

T1	T2
$l_1(A); r_1(A);$	
	$l_2(B); r_2(B);$
A:=A+100;	
	B:=B*2;
$w_1(A);$	
	$w_2(B);$
$l_1(B);$ Denied	$l_2(A);$ Denied

- **Neither transaction can proceed, and they wait forever.**

Shared and Exclusive locks

- Shared lock (S) and Exclusive (X) lock
  - Shared lock
    - SI_i (X): transaction Ti requests a shared lock on database element X
  - Exclusive lock
    - XI_i (X): transaction Ti requests an exclusive lock on database element X
  - Ui(X): Ti unlocks X or T relinquishes whatever lock(s) it has on X.

Shared/ Exclusive lock system

1. Consistency of transactions
   - A read action ri(X) must be preceded by sli(X) or xli(X), with no intervening ui(X).
   - A write action wi(X) must be preceded by xli(X), with no intervening ui(X).
   - All locks must be followed by an unlock of the same element.
2. Two-phase locking of transactions
   - Locking must precede unlocking, or no action sli(X) or xli(X) can be preceded by an action ui(Y), for any Y.
3. Legality of schedule
   - An element may either be locked exclusively by one transaction or by several in shared mode, but not both.
   - If $xl_i(X)$ appears in a schedule, then there cannot be a following $xl_j(X)$ or $sl_j(X)$ for some j other than i, $j \neq i$ without an intervening $u_i(X)$.
   - If $sl_i(X)$ appears in a schedule, then there cannot be a following $xl_j(X)$, $j \neq i$ without an intervening $u_i(X)$.

Compatibility Matrices

	S	X
S	yes	no
X	no	no

Example:

T1	T2
$sl_1(A);r_1(A);$	
	$sl_2(A);r_2(A);$
	$sl_2(B);r_2(B);$
$xl_1(B)$ denied	
	$u_2(A);u_2(B);$
$xl_1(B);\ r_1(B);\ w_1(B);$	
$u_1(A);u_1(B);$	

The schedule is conflict-serializable. The conflict-equivalent serial order is T2< T1.

Upgrading locks

- Transaction T that wants to read and write a new value of X
  - First, T takes a shared lock on X
  - Later, when T is ready to write the new value, upgrade the lock to exclusive.
- When T takes a shared lock on X, other transactions are allowed to read X at the same time.

Example

T1	T2
$sl_1(A);r_1(A);$	
	$sl_2(A);r_2(A);$
	$sl_2(B);r_2(B);$
$sl_1(B);r_1(B);$...	
$xl_1(B)$ denied	
	$u_2(A);u_2(B);$
$xl_1(B);\ w_1(B);$	
$u_1(A);u_1(B);$	

If T1 had asked for an exclusive lock on B initially, before reading B, then the request would have been denied, because T2 already had a shared lock on B.

Upgrading lock allows more concurrent operation

Upgrading & Deadlock

T1	T2
sl_1(A);	
	sl_2(A);
xl_1(A) denied	
	xl_2(A); denied

Update Lock $ul_i(X)$

- Can avoid the deadlock problem.
- An update lock uli(X) gives transaction Ii only the privilege to read X, not to write X.
- Only the update lock can be upgraded to write lock later.
  - A read lock cannot be upgraded.

Compatible Matrix

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

We can grant an update lock on X when there are already shared locks on X, but once there is an update lock on X, we prevent additional locks of any kind from being taken on X. The reason is that, if we don't deny such locks, the the updater may never get a chance to upgrade to exclusive, since there would always be other locks on X

Example

T1	T2
$ul_1(A)$, $r_1(A)$;	
	$ul_2(A)$, Denied
$xl_1(A)$, $w_1(A)$, $u_1(A)$;	
	$ul_2(A)$, $r_2(A)$;
	$xl_2(A)$, $w_2(A)$, $u_2(A)$;

Increment lock

- Increment lock $il_i(X)$ $inc_i(X)$
  - Increment actions commute with each other: if 2 transactions add constants to the same database element, it does not matter which goes first.
    - Incrementation commutes with neither reading nor writing.
  - Use INC(A,c) to stand for the atomic execution of the steps:

    read (A,t); t:=t+/-c; write(A,t);

Shared/ Exclusive/ Increment lock system

- Consistency of transaction

  A consistent transaction can only have an increment action on X if it holds an increment lock on X at the time. An increment lock does not enable either read nor write actions

- Legality of schedule
  - Many transactions can hold an increment lock on X at any time.
  - If T holds an increment lock on X, no other transaction can hold shared lock or exclusive lock on X at the same time.
  - Compatibility matrix:

	S	X	I
S	Yes	No	No
X	No	No	No
I	No	No	Yes

Example

T1	T2
$sl_1(A)$, $r_1(A)$;	
	$sl_2(A)$, $r_2(A)$;
	$il_2(B)$, $inc_2(B)$;
$il_1(B)$, $inc_1(B)$;	
	$u_2(A)$, $u_2(B)$;
$u_1(A)$, $u_1(B)$;	

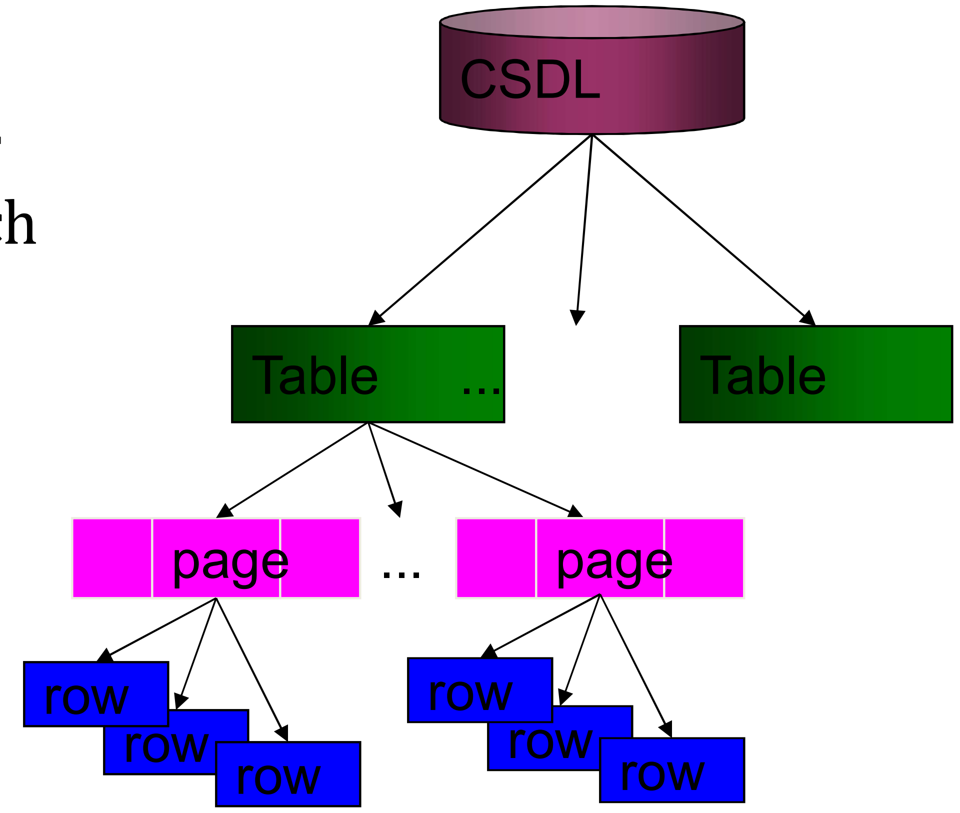
Locks with multiple granularity

- Database elements can be: tuple, page/ block, relation.
- Some apps profit from small db elements, such as tuples, while others are bests off with large elements.
- In bank database:
  - Some transactions need exclusive locks on individual account tuples.
  - Some transactions need shared lock on the account relation (for computing some aggregation).
  - ➔ leading to unserializable behavior.

- For managing locks at different granularities, we need a new kind of lock called “warning”.
- Warning locks are useful when the database elements form a nested or hierarchical structure.

3 levels of database elements

- Relations are the largest lockable elements.
- Each table consists of one or more blocks/pages (on which tuples are stored).
- Each block contains one or more tuples/ rows.



Warning lock

- *Intent Locks*
  - Will be denoted by prefixing I to the ordinary locks:

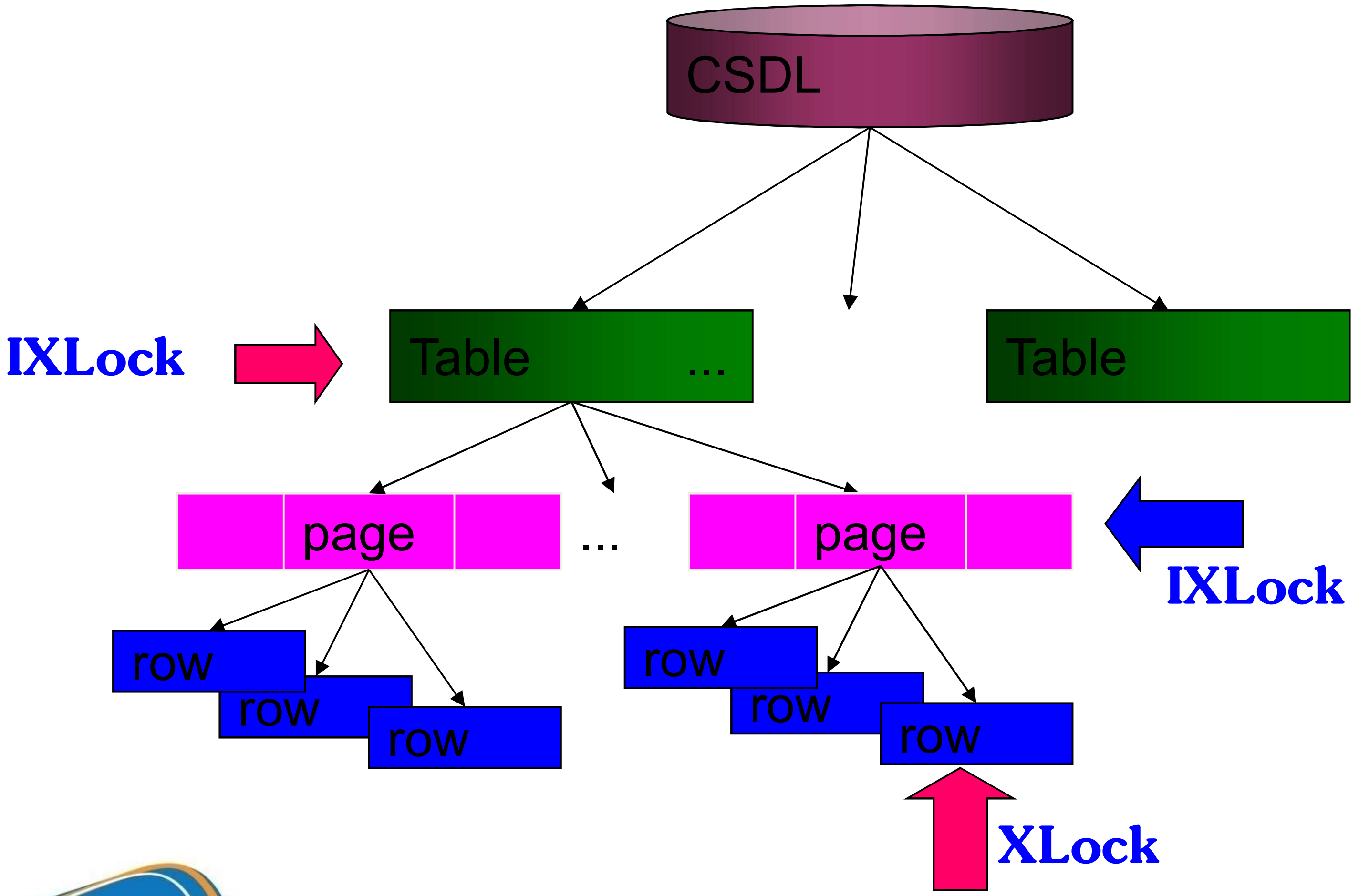
Shared Lock	=> Intent Shared Lock	(IS)
Update Lock	=> Intent Update Lock	(IU)
Exclusive Lock	=> Intent Exclusive Lock	(IX)

Warning protocol

1. To place an ordinary S or X lock on any element, we must begin at the root of the hierarchy.
2. If we are at the element that we want to lock, we need look no further. We request an S or X lock on that element.
3. If the element we wish to lock is further down the hierarchy, then we place a warning at this node.
   - If we want to get a shared lock (exclusive lock) on a subelement we request an IS lock (IS lock) on this node.
4. When the lock on the current node is granted, we proceed to the appropriate child.
5. Repeat step 2 or 3.

Warning protocol

- *Intent Locks*



Compatibility matrix

	IS	IX	S	X
IS	Yes	Yes	Yes	No
IX	Yes	Yes	No	No
S	Yes	No	Yes	No
X	No	No	No	No

Precedence graph

- For testing the serializability of a schedule with locks.
- Input: S: T1, T2, ..., Tk.
- Output: Serializable/ Unserializable.
- All the precedence in S can be summarized in a precedence graph.
  - The nodes of the precedence graph are the transactions of a schedule S. Each node is labeled i for the transaction Ti.
  - There is an arc from node i to node j in the following cases:
    - If Ti requests Rlock on X, $Rlock_i(X)$, Tj requests Xlock on X at the same time, $Xlock_j$ (X), and $Rlock_i(X)$ is ahead $Xlock_j$ (X) in S or
    - If Ti requests Xlock on X, $Xlock_i(X)$, Tj requests Xlock on X at the same time, $Xlock_j$ (X), and $Xlock_i(X)$ is ahead $Xlock_j$ (X) in S
- If there are any cycles in precedence graph for S, S is not conflict-serializable.
- If the precedence graph is acyclic, S is conflict-serializable. Any topological of the nodes is a conflict-equivalent serial order.

Example

T1	T2
RL(A)	
Read(A)	
UL(A)	
	RL(B)
	Read(B)
	UL(B)
	WL(A)
	Read(A)
	A:=A+B
	Write(A)
	UL(A)
WL(B)	
Read(B)	
B:=B+A	
Write(B)	
UL(B)	

S1 $$\begin{tikzcd} T_1 & & T_2 \end{tikzcd}$$

S2 $$\begin{tikzcd} T_1 \arrow[rr, bend left] & & T_2 \arrow[ll, bend left] \end{tikzcd}$$ G

S3 There is a cycle in G => S is not conflict-serializable.

Motivation for tree-based locking

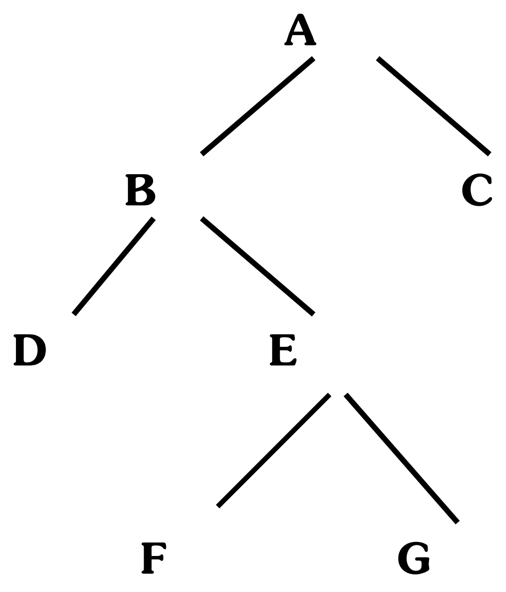
- Tree of elements, formed by the nesting structure of the db elements: the children being subparts of the parent.
- If 2PL is used, the concurrent use of B-tree (index) is impossible.
  - Every transaction using the index must begin by locking the root node of the B-tree.
  - If the transaction is 2PL, it cannot unlock the root until it has acquired all the locks it needs.
  - Only one transaction that is not read-only can access the B-tree at any time.

Rules for access to tree-structured data

1. A transaction's first lock may be at any node of the tree.
2. Subsequent locks may only be acquired if the transaction currently has a lock on the parent node.
3. Nodes may be unlocked at any time.
4. A transaction may not relock a node on which it has released a lock, even if it still holds a lock on the node's parent.

- The tree protocol forces a serial order on the transactions involved in a schedule.

Example

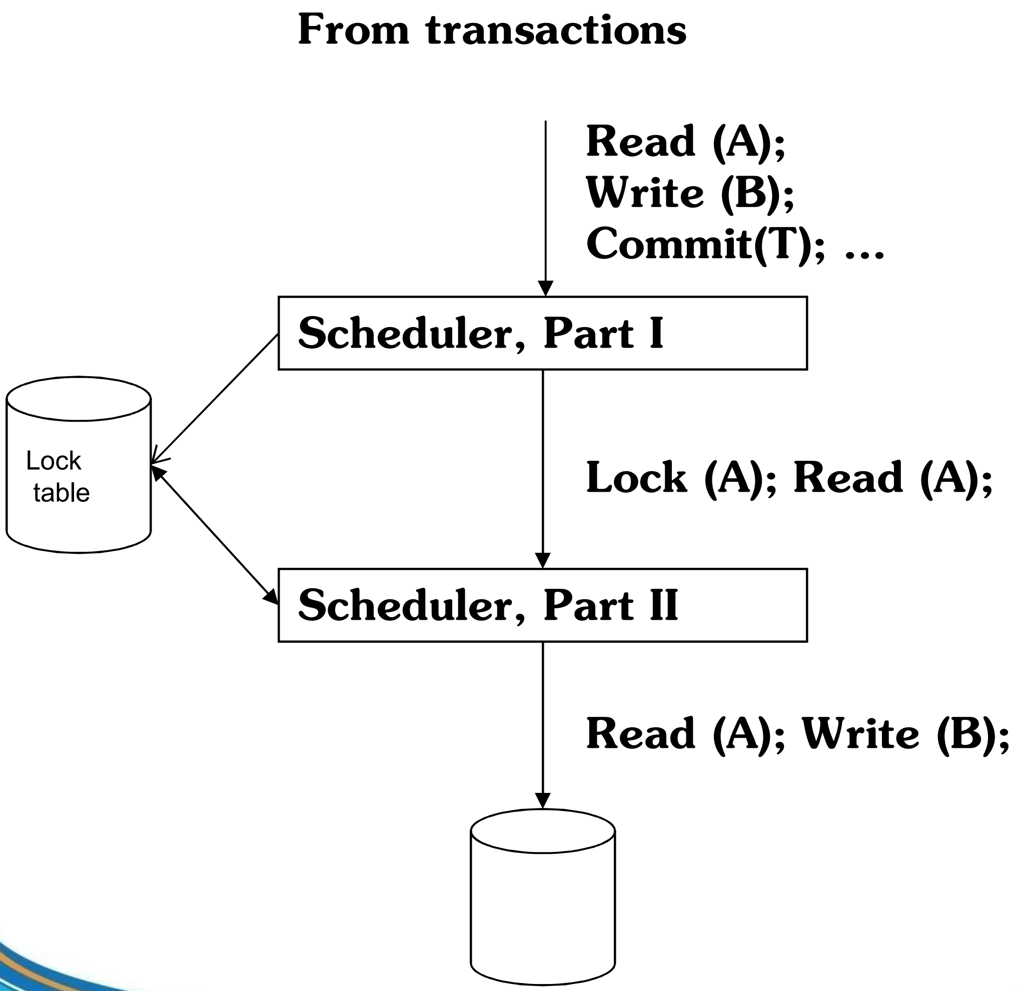


T1	T2	T3
l1(A), r1(A); l1(B), r1(B); l1(C), r1(C) w1(A), u1(A); l1(D), r1(D) w1(B), u1(B)		
	l2(B), r2(B);	
		l3(E), r3(E);
w1(D), u1(D);		
w1(C), u1(C);		
	l2(E), Chờ	
		l3(F), r3(F) w3(F), u3(F);
		l3(G), r3(G) w3(E), u3(E);
	l2(E), r2(E)	
		w3(G), u3(G);
	w2(B), u2(B); w2(E), u2(E);	

An architecture for a locking scheduler

- The transactions themselves do not request locks. It is the job of the scheduler to insert lock actions into the stream of reads, writes, and other actions that access data.
- Transactions do not release locks. The scheduler release locks when the transaction manager tells that the transaction will commit or abort.

Scheduler's job



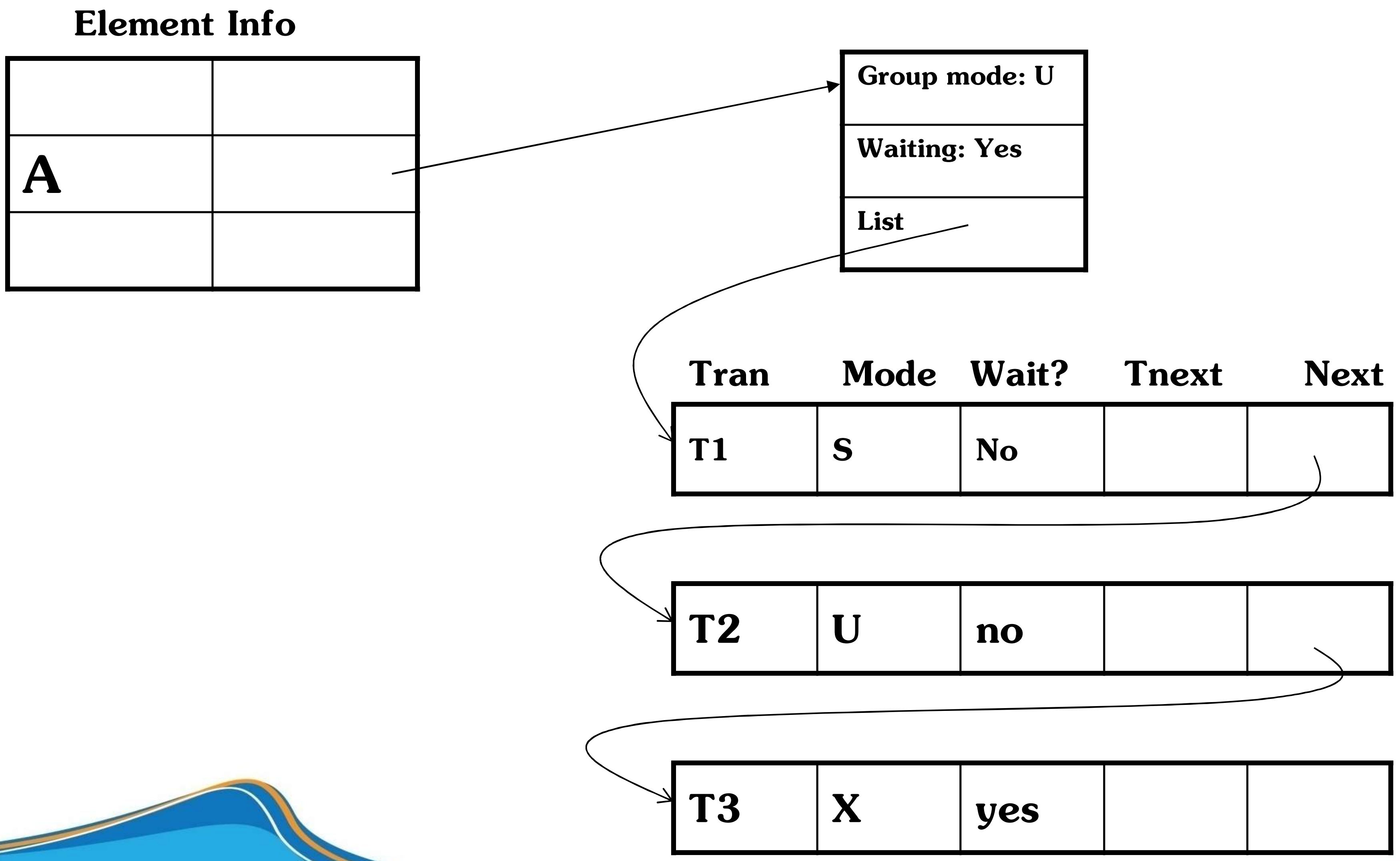
How the scheduler works?

1. Part I: takes the stream of requests generated by the transactions and inserts appropriate lock actions ahead of all the database access operations.
2. Part II: takes the sequence of lock and database-access actions passed to it by Part, and executes each appropriately.

- T is delayed: a lock has not been granted, the action is added to a list of actions must be performed for T.
- T not delayed:
  - If the action is a database access, it is transmitted to the database and executed.
  - If it is a lock action, it examines the lock table to see if the lock can be granted.
    - if so, the lock table is modified to include the lock just granted.
  - If not, an entry must be made in the lock table to indicate that the lock has been requested. Part II delays further actions for transaction T, until the time the lock is granted.

3. When a transaction T commits or aborts, Part I is notified by the transaction manager, and releases all locks held by T. If any transactions are waiting for any of these locks, Part I notifies Part II.
4. When Part II is notified that a lock on some db element X is available, it determines the next transaction or transactions that can now be given a lock on X.

Lock table


Element Info
A
Group mode: U
Waiting: Yes
List
Tran
Mode
Wait?
Tnext
Next
T1
S
No
T2
U
no
T3
X
yes

The lock table

- The lock table is a relation that associates db elements with lock information about that element.
- Group mode: is a summary of the most stringent conditions that a transaction requesting a new lock on A.
  - Is “S” if only shared locks are held.
  - Is “U” if there is one update lock and perhaps one or more shared locks.
  - Is “X” if there is one exclusive lock and no other locks.
- Waiting bit: tells that there is at least o transaction waiting for a lock on A.
- List: consists of all the transactions that either currently hold locks on A or are waiting lock on A:
  - The name of the transaction holding/ waiting for a lock.
  - The mode of the lock.
  - Whether the transaction is holding or waiting for the lock.

Handling lock requests/ unlocks

- T requests a lock on A:

  If there is no lock-table entry for A, the entry is created and the request is granted.

  Else

  use the group mode to decide whether or not the lock is granted.

- T unlocks A
  - Delete T’s entry on the list.
  - Modified the group mode.
  - Grant the lock(s) from the list of requested locks.
    - First-come-first-served: no starvation.
    - Priority to shared lock: first grant all the shared locks waiting. Then grant one update lock, if there are any waiting. Only grant an exclusive lock if no others are waiting. (may lead to starvation).
    - Priority to updating: first grant to transaction with a U lock waiting to upgrading to an X lock. Otherwise, follow other strategies.

6. ISOLATION LEVELS

Isolation levels

- Executions of transactions follow ACID properties.
- Isolation determines how transaction integrity is visible to other users and systems. It means that a transaction should take place in a system in such a way that it is the only transaction that is accessing the resources in a database system.
- Isolation levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system.
- The SQL standard defines four isolation levels :
  - Read Uncommitted.
  - Read Committed (default).
  - Repeatable Read.
  - Serializable.

Isolation levels

- Each transaction works under an isolation level.
- T-SQL command for setting transaction's isolation level:

```
SET TRANSACTION ISOLATION LEVEL <READ COMMITTED|READ
UNCOMMITTED|REPEATABLE READ|SERIALIZABLE>
```

Read Uncommitted

- Read Uncommitted is the lowest isolation level.
- The transaction doesn’t request any read lock on the row it want to read.
- However, the transaction requests the write lock on the row to be written.
- In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads.
- In this level, transactions are not isolated from each other.

Read Committed

This isolation level guarantees that any data read is committed at the moment it is read.

Thus it does not allows dirty read.

The transaction holds a read or write lock on the current row, and thus prevent other transactions from reading, updating or deleting it.

Repeatable Read

The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes.

So other transactions cannot read, update or delete these rows.

Serializable

This is the Highest isolation level.

A *serializable* execution is guaranteed to be serializable.

Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

Examining the isolation levels

- Connect to SQL Query Analyzer 2 times to make 2 connections concurrently.
- T1 uses the following command for waiting the execution of T2:

```
        WAITFOR DELAY 'hh:mm:ss'
```

- Ex: SINHVIEN (<u>MASV</u>, TEN)

MASV	**TEN**
1	**Nam**
2	**Toan**
3	**Tam**

TH1: a. Read UnCommitted & Read Committed

T1	T2

BEGIN TRAN
UPDATE *SINHVIEN*
SET *TEN* = 'Minh'
WAITFOR DELAY '*00:00:20*'	

	BEGIN TRAN
	SET TRANSACTION ISOLATION LEVEL ***READ UNCOMMITTED***
	SELECT * FROM *SINHVIEN* WHERE *TEN* = 'Minh'
	COMMIT TRAN

ROLLBACK TRAN	

T2 returns all the rows in table SINHVIEN

TH1: b. Read UnCommitted & Read Committed

T1	T2

UPDATE *SINHVIEN*
SET *TEN* = 'Minh'
WAITFOR DELAY '*00:00:20*'	

	BEGIN TRAN
	SET TRANSACTION ISOLATION LEVEL *READ COMMITTED*
	SELECT * FROM *SINHVIEN* WHERE *TEN* = 'Minh'
	COMMIT TRAN

ROLLBACK TRAN	

T2 returns no rows.

TH2: a. Read Committed & Repeatable Read

T1	T2
BEGIN TRAN	
	BEGIN TRAN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED	
SELECT TEN FROM SINHVIEN	
WAITFOR DELAY '00:00:20'	
	UPDATE SINHVIEN
	SET TEN= 'Minh'
	COMMIT TRAN
SELECT TEN FROM SINHVIEN	
COMMIT TRAN	

Results returned by 2 selections of T1 are different.

TH2: a. Read Committed & Repeatable Read

T1	T2
BEGIN TRAN	
	BEGIN TRAN
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ	
SELECT TEN FROM SINHVIEN	
WAITFOR DELAY ‘00:00:20’	
	UPDATE SINHVIEN
	SET TEN= ’Minh’
	COMMIT TRAN
SELECT TEN FROM SINHVIEN	
COMMIT TRAN	

Results of 2 selections of T1 are the same.

TH3: a. Repeatable Read & Serializable

T1	T2
BEGIN TRAN	
	BEGIN TRAN
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ	
SELECT TEN FROM SINHVIEN	
WAITFOR DELAY ‘00:00:20’	
	INSERT INTO SINHVIEN VALUES (‘4’,‘Tuyet’)
	COMMIT TRAN
SELECT TEN FROM SINHVIEN	
COMMIT TRAN	

Results returned by 2 selections of T1 are different.

TH3: a. Repeatable Read & Serializable

T1	T2
BEGIN TRAN	
	BEGIN TRAN
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE	
SELECT TEN FROM SINHVIEN	
WAITFOR DELAY '00:00:20'	
	INSERT INTO SINHVIEN VALUES ('4','Tuyet')
	COMMIT TRAN
SELECT TEN FROM SINHVIEN	
COMMIT TRAN	

Results returned by 2 selections of T1 are the same.

Summarization

Read Uncommitted (MCL1)	Read committed (MCL2)	Repeatable read (MCL3)	Serializable (MCL4)
1. Write lock? 2. Read lock? 3. Can avoid what phenomena of concurrency control?	1. Write lock? 2. Read lock? 3. Can avoid what phenomena of concurrency control?	1. Write lock? 2. Read lock? 3. Can avoid what phenomena of concurrency control?	1. Write lock? 2. Read lock? 3. Can avoid what phenomena of concurrency control?
3.1 Lost Update	6.1 Lost Update	9.1 Lost Update	12.1 Lost Update
3.2 Dirty read	6.2 Dirty read	9.2 Dirty read	12.2 Dirty read
3.3 Unrepeatable read	6.3 Unrepeatable read	9.3 Unrepeatable read	12.3 Unrepeatable read
3.4 Phantom	6.4 Phantom	9.4 Phantom	12.4 Phantom

7. DEADLOCK

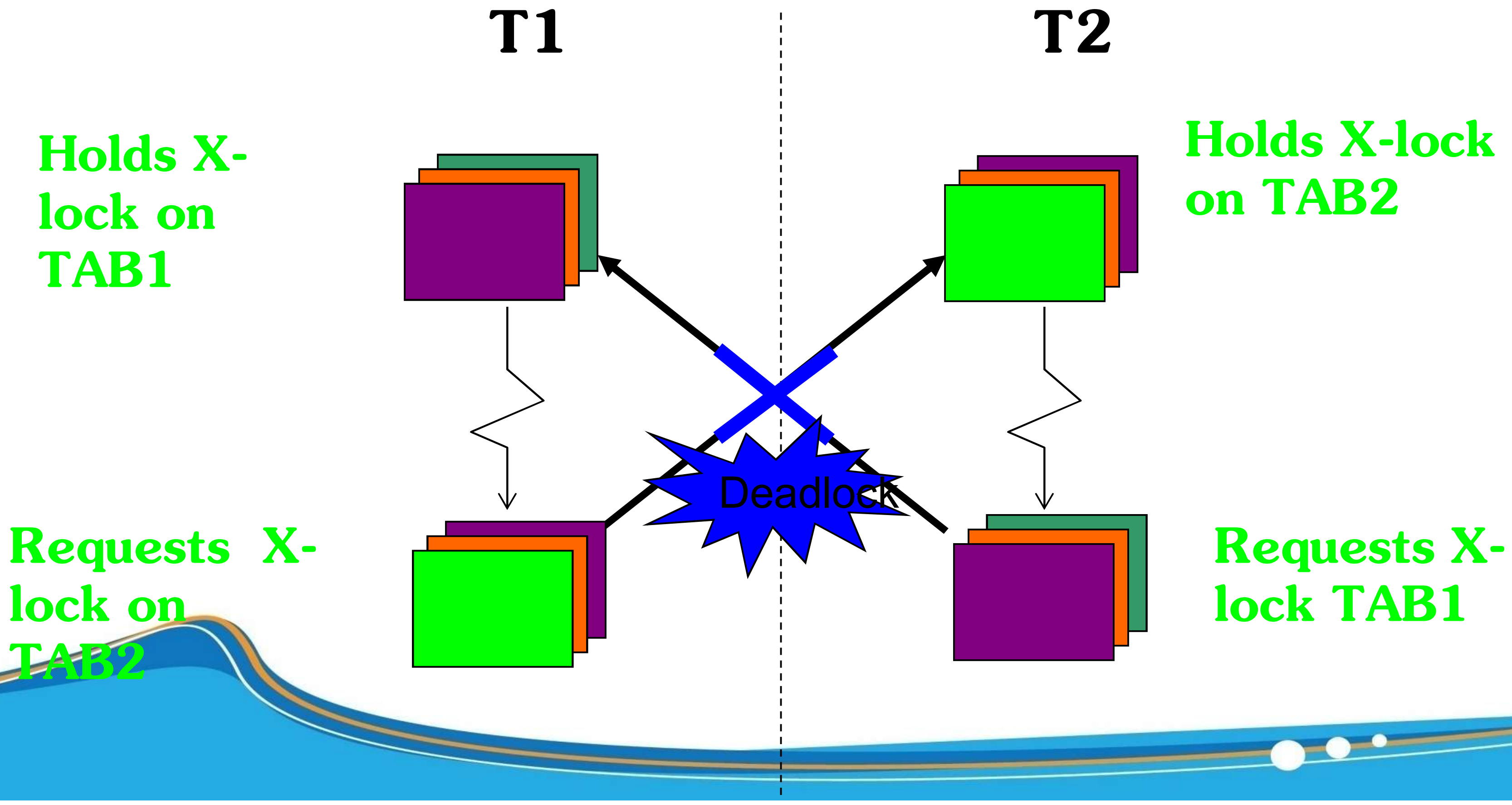
- Concepts
- Deadlock detection
- Deadlock prevention

Deadlock

- Concurrently executing transactions can compete resources and reach a state called deadlock.
- Deadlock is a state in which each of several transactions is waiting for a resource held by one of the others, and none can make progress.
- There are to cases of deadlock:
  - Cycle deadlock
  - Conversion deadlock

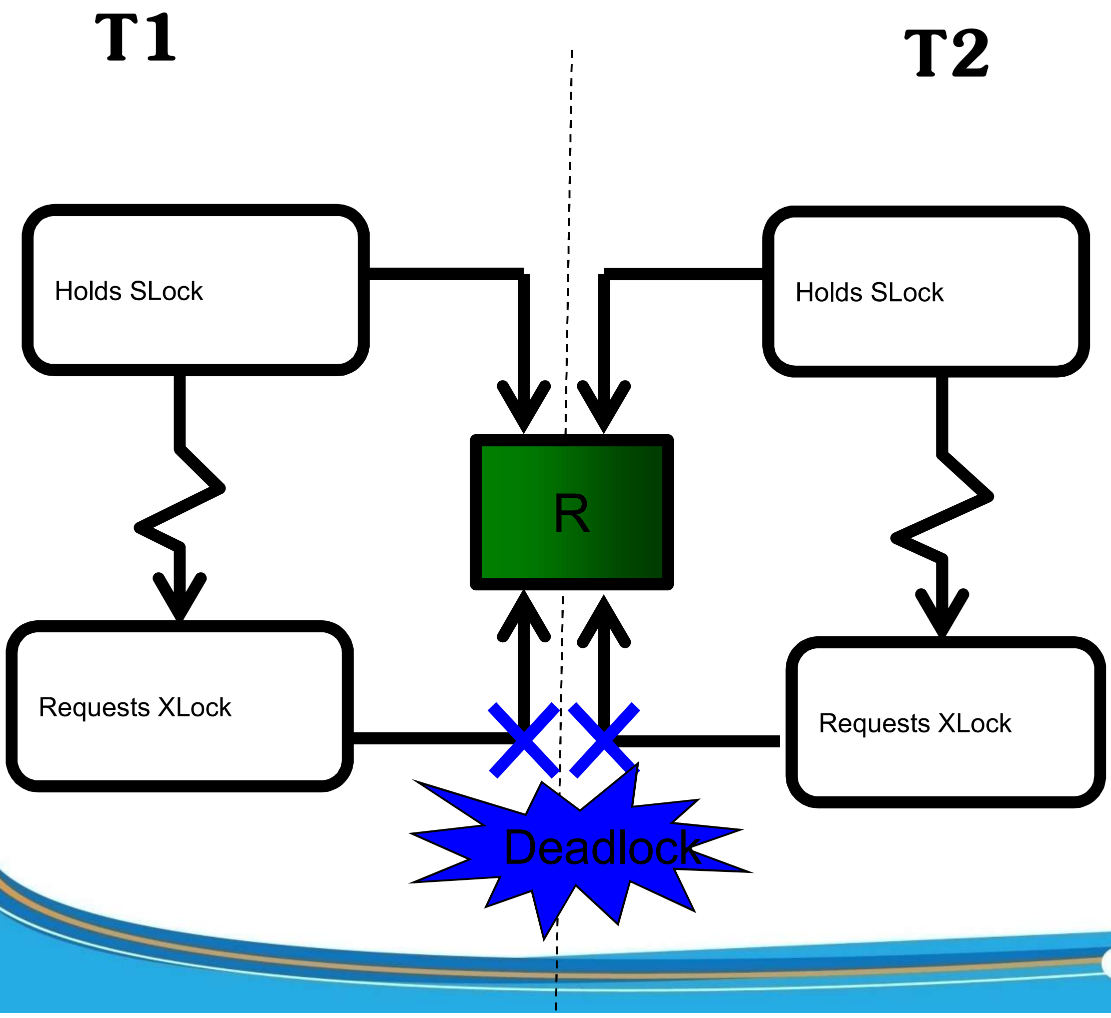
Cycle deadlock

- T1 holds X-lock on TAB1, T2 holds X-lock on TAB2
- T1 requests X-Lock on TAB2 => T1 is waiting for T2
- T2 requests X-Lock on TAB1 => T2 is waiting for T1



Conversion Deadlock

Both T1 and T2 hold S-Lock on the resource R
T1 requests X-Lock on R => T1 waiting for T2
T2 requests X-Lock on R => T2 waiting for T1



Deadlock

- Deadlock detection by timeout
  - When a deadlock exists, it is impossible to repair the situation so that all transactions involved can proceed.
  - At least one of the transactions will have to be rolled back (aborted and restarted).
  - Detect and resolve deadlocks with a *timeout*.
    - A limit on how long a transaction may be active.
    - If a transaction exceeds this time, roll it back.
    - In simple system: transactions execute in milliseconds → timeout may be in minute.
    - This is not convenient in some cases.

The Waits-For Graph

- Waits-for graph is used for indicating which transactions are waiting for locks held by another transaction.
- The waits-for graph has a node for each transaction that currently holds a lock or is waiting for one.
- There is an arc from node (transaction) T to node U if:
  - U holds a lock on A.
  - T is waiting for a lock on A and
  - T cannot get a lock on A in its desired mode unless U first releases its lock on A.
  - If there are no cycles in the waits-for graph, each transaction can eventually complete.
  - If there is a cycle, no transaction in the cycle can ever make progress (so there is a deadlock). A strategy for deadlock avoidance is to roll back any transaction that makes a request that would cause a cycle in the waits-for graph.

- Waits – for graph can be large and time-consuming for analyzing it.

Example

T1	T2	T3
Rlock(A)		
	Rlock(C)	
		Wlock(E)
Wlock (B)		
	Rlock(B)	
		Rlock(B)
Wlock(C)		
	Wlock(E)	
	Rlock(D)	
		Wlock(C)
…	…	…

Is there a deadlock?

Deadlock Prevention

- By Ordering Elements
  - Order db elements in some arbitrary but fixed order.
  - If every transaction is required to request locks on elements in order, there can be no deadlock due to transactions waiting for locks.

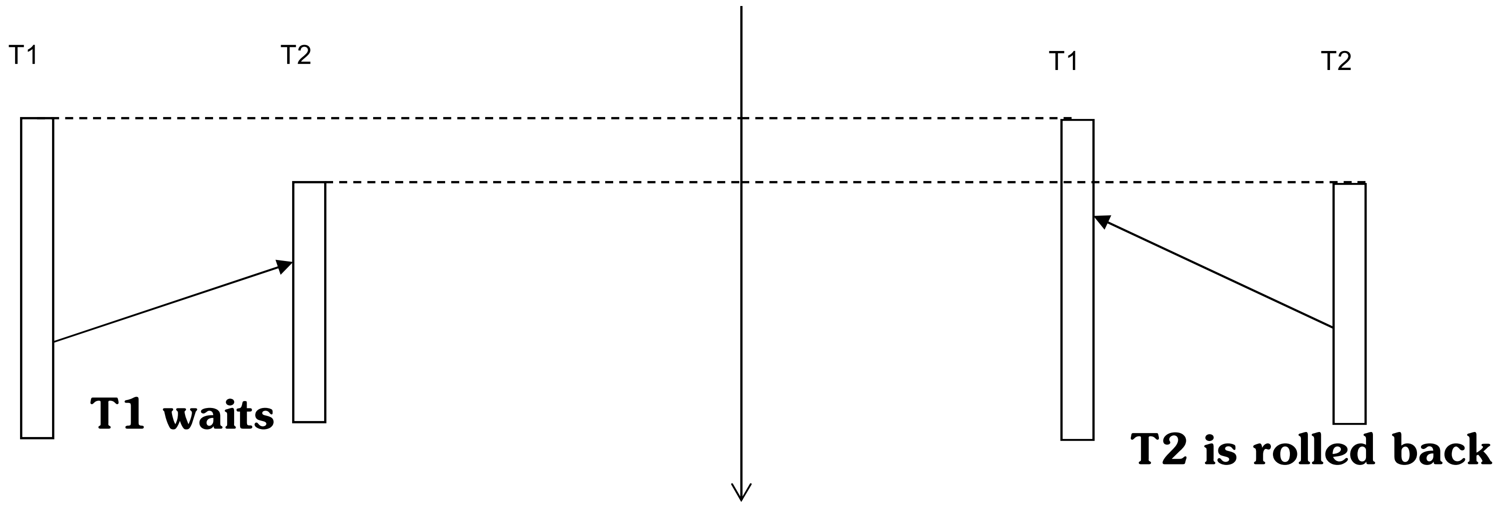
Detecting deadlocks by timestamps

- Associate with each transaction a timestamp. This timestamp:
  - Is for deadlock detection only. (It is not the same as the timestamp used for concurrency control.)
  - If a transaction is rolled back, it restarts with a new, later concurrency timestamp, but its timestamp for deadlock detection never changes.
- The timestamp is used when a transaction Ti has to wait for a lock that is held by another transaction Tj.

Wait - Die Scheme

Ti, Tj have timestamps t_{Ti},t_{Tj}, respectively. Ti waits for a lock that is held by Tj

```
If $t_{Ti} < t_{Tj}$ then (Ti is older than Tj)
    Ti is allowed to wait for the lock(s) held by Tj
Else (Tj is older than Ti)
    Ti "dies"; it is rolled back Ti
EndIf
```



Wound – Wait Scheme

Ti, Tj have timestamps t_{Ti},t_{Tj}, respectively. Ti waits for a lock that is held by Tj

If $t_{Ti} < t_{Tj}$ then (Ti is older than Tj)

Ti wounds Tj. Tj must roll back and relingquish to Ti the lock(s) that Ti needs from Tj

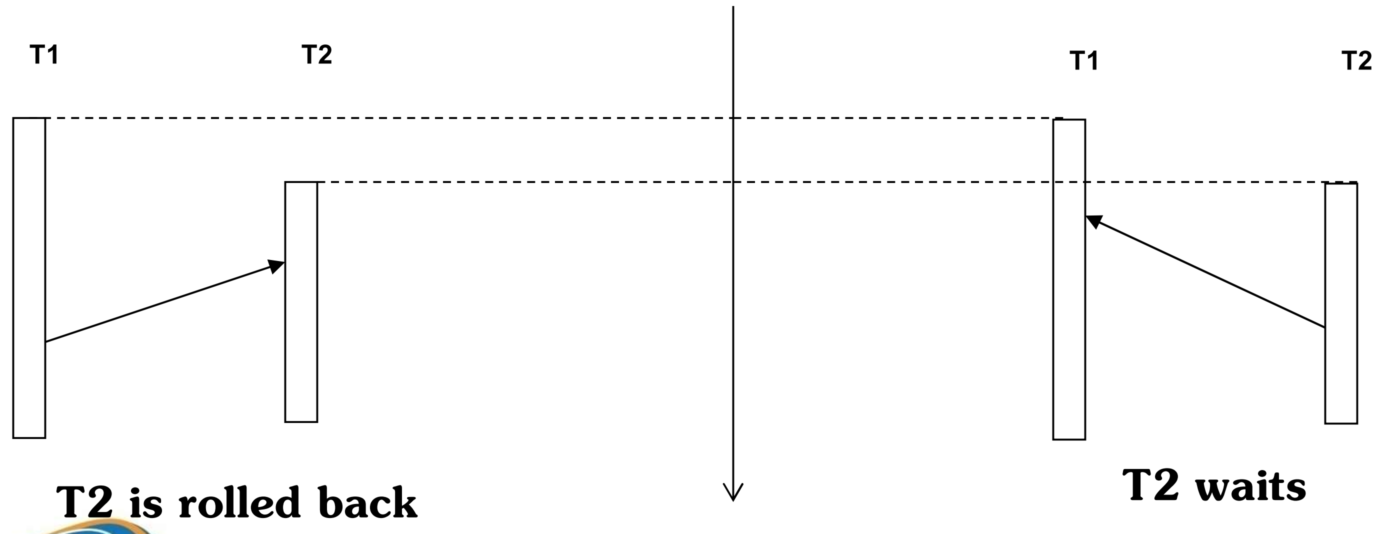
Exception:

At the time the "wound" takes effect, Tj has already finished and released its locks. Tj survives and need not be rolled back.

Else (Tj is older than Ti)

Ti waits for the lock(s) held by Tj

EndIf



8. HOW TO USE THE LOCK MODES

Lock modes

- Lock modes provided by SQL Server:
  - FASTFIRSTROW
  - HOLDLOCK
  - NOLOCK
  - PAGLOCK
  - READCOMMITTED
  - READPAST
  - READUNCOMMITTED
  - REPEATABLEREAD
  - ROWLOCK
  - SERIALIZABLE
  - TABLOCK
  - TABLOCKX
  - UPDLOCK
  - XLOCK

Example

- `SELECT COUNT(*) FROM SINHVIEN WITH (TABLOCK, HOLDLOCK)`

- We can combine multiple lock modes at the same time:

`(TABLOCK, XLOCK)` : for holding exclusive lock on the (given) table.

`(ROWLOCK, XLOCK)` : for holding exclusive lock on the row.

Example

GP (STT, NGAY CAP, SOXE, LYDOCAP)

Suppose that STT is 10 currently.

Each value of STT is an integer, consecutively.

Steps	Transaction CGP
S1	Begin Tran
S2	M= Select STT from GP where STT = (Select max (STT) from GP)
S3	Insert into GP values (M+1,…)
S4	Commit tran

U1 calls CGP	U2 calls CGP
S1	
	S1
S2 10	
	S2 10
S3 11	
	S3 Loãi 11
S4	
	~~S4~~

Example

GP (<u>STT</u>, NGAY CAP, SOXE, LYDOCAP)

Steps	Transaction CGP
S1	Begin Tran
S2	M= Select STT from GP with (rowlock, xlock) where STT = (Select max (STT) from GP)
S3	Insert into GP values (M+1,...)
S4	Commit tran

U1 calls CGP	U2 calls CGP
S1	
	S1
S2 10	
	S2 U2 waits
S3 11	
S4	
	S2 11
	S3 ✓ 12
	S4

9. Concurrency Control by Timestamps

Timestamp of transactions

- The scheduler assigns to each transaction T a unique number, its timestamp TS(T).
  - Timestamps must be issued in ascending order, at the time the transaction starts.
  - Timestamps may be based on the system clock or a counter.

Timestamp of database elements

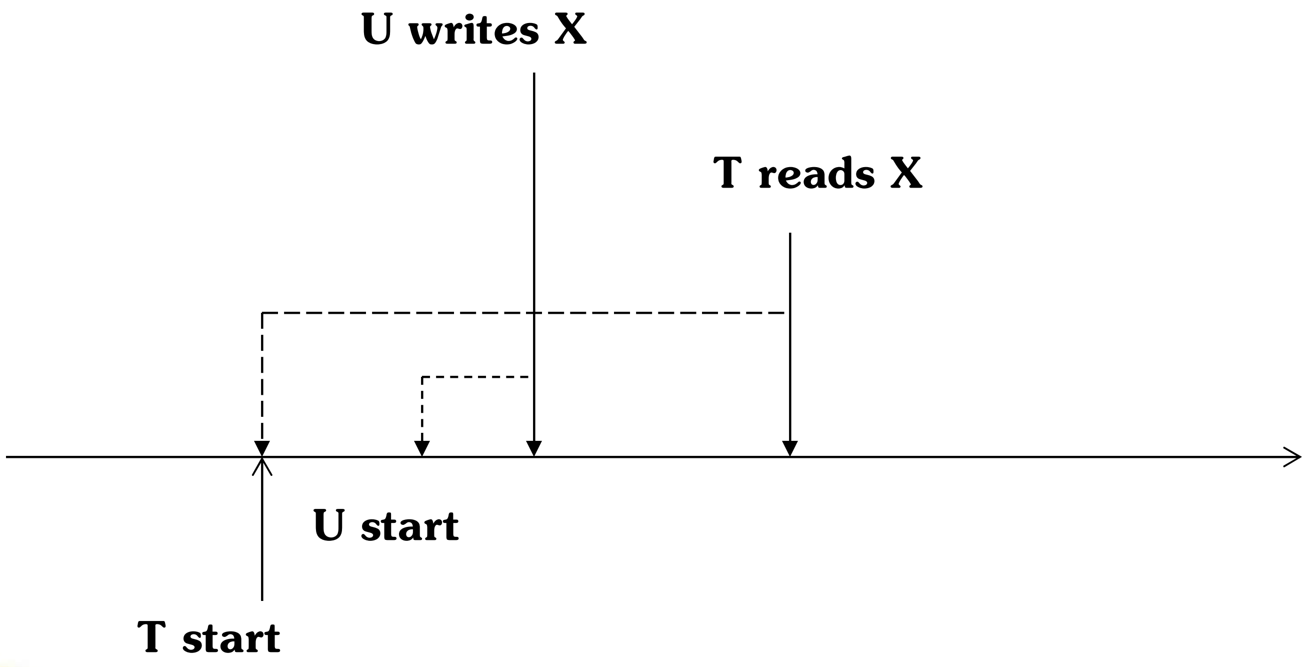
- Associates with each database element X two timestamps and an additional bit:
  - RT(X): the read time of X, which is the highest timestamp of a transaction that has read X.
  - WT(X): the write time of X, which is the highest timestamp of a transaction that has written X.
  - C(X); the commit bit for X, which is true if and only if the most recent transaction to write X has already committed.

Concepts

- The timestamp-based scheduler assures that the serial schedule according to the transactions’ timestamps is equivalent to the actual schedule of the transactions.
  - If TS(Ti) < TS(Tj), then in the equivalent serial schedule, Ti<Tj.

Read too late

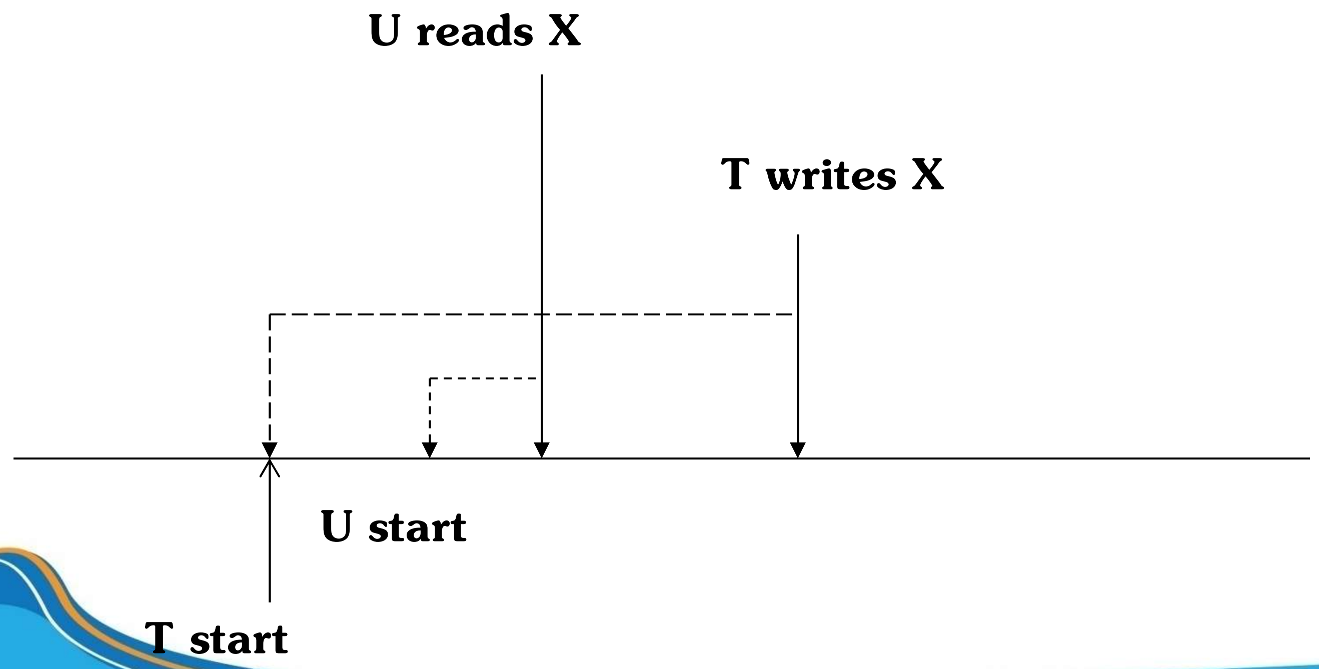
- T tries to read database element X
- However, TS(T) < WT(X)=TS(U) : X was written by U before T executed.
- Abort T.



Write too late

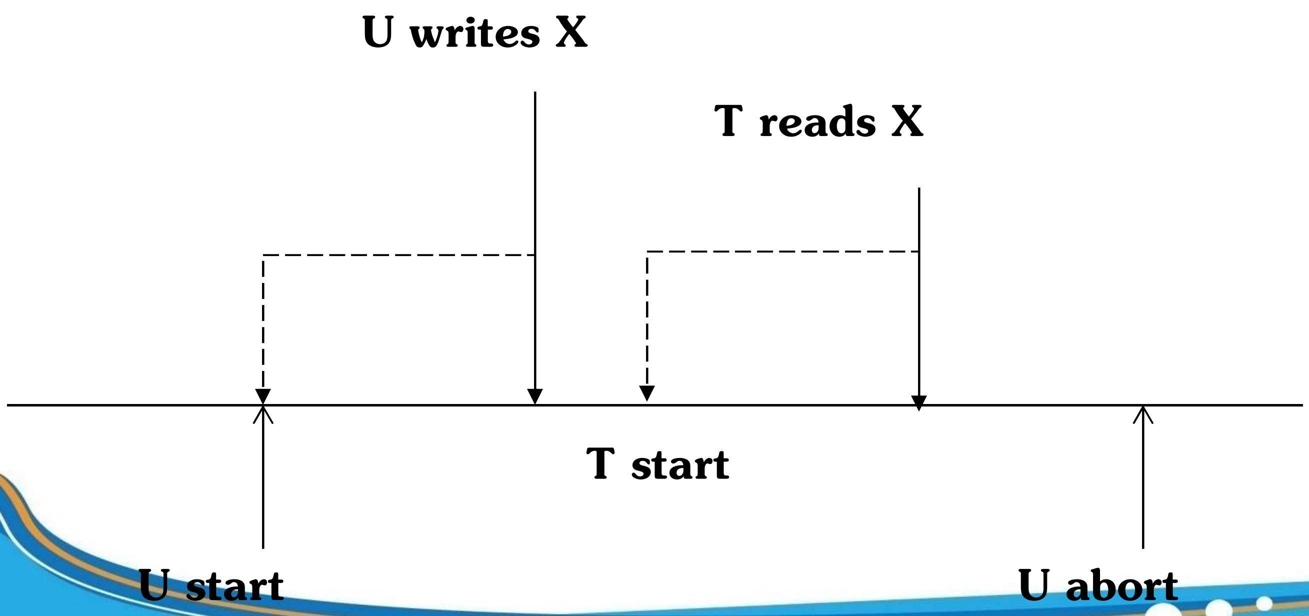
- T tries to write X.
- However, read time of X indicates that some other transaction that should have read the value written by T but read some other value instead:

$$WT(X) < TS(T) < RT(X$$



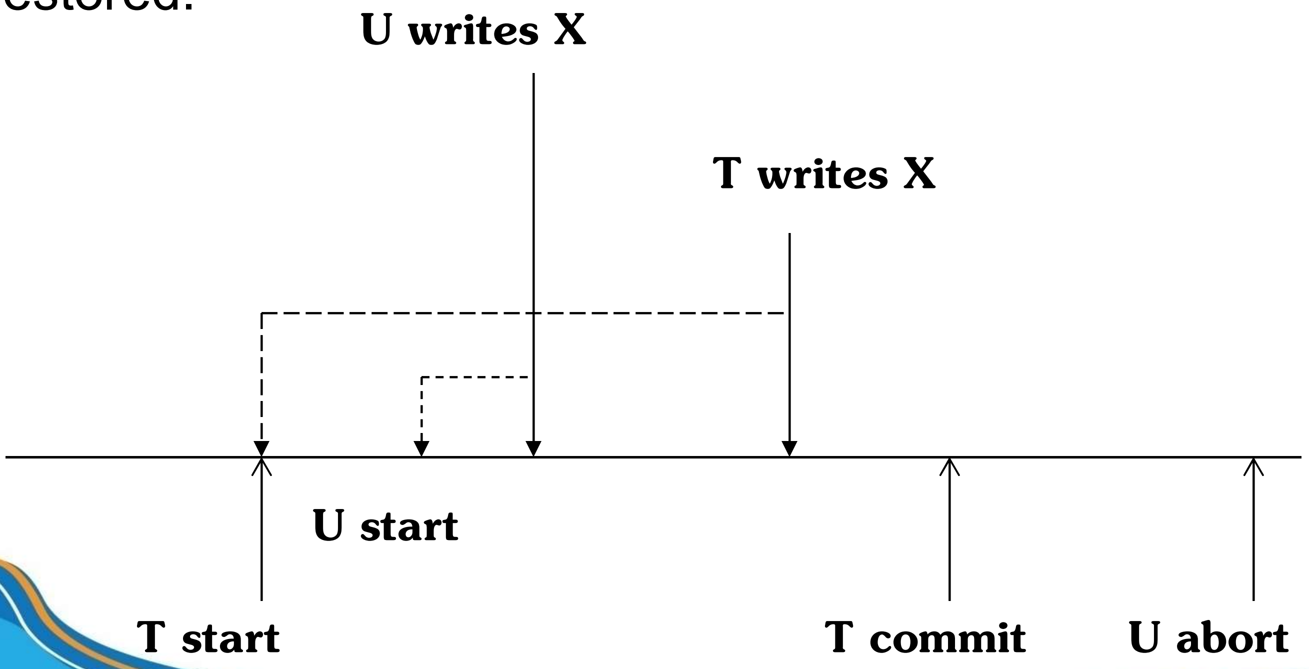
Dirty read

- T reads X, X was last written by U. However, it is possible that after T reads the value of X written by U, U will abort.
  - Delay T'read until U commits or aborts (based on the value of commit bit C(X)).



Dirty read

- U is the transaction with a later timestamp than T, has written X first.
- When T tries to write, the appropriate action is to do nothing (Thomas write rule)
- if U later aborts and T commit
  - The value of X should be the one written by T, however, skipped.
  - The value of X should be removed, the previous value and write-time restored.



Rules

- T requests a read or a write, in response, the scheduler:
  - Granting the request.
  - Aborting T and restarting T with a new timestamp, or
  - Delaying T and later deciding whether to abort T or to grant the request.

Timestamp-based CC

1. T requests a Read on X
   - If TS(T) >= WT(X)
     - If C(X) = TRUE, grant the request.
       - If TS(T) > RT(X)
         - RT(X) := TS(T)
       - Otherwise do not change RT(X)
     - If C(X) = FALSE, delay T until C(X) becomes true or the transaction that wrote X aborts.
   - If TS(T) < WT(X)
     - Rollback T (Read too late), restart T with a new, larger timestamp.

Timestamp-based CC

2. T requests a Write on X
   - If TS(T) >= RT(X) and TS(T) >= WT(X)
     - Write a new value for X
     - Set WT(X) := TS(T).
     - Set C(X) := FALSE.
   - Neáu TS(T) >= RT(X) and TS(T) < WT(X)
     - If C(X) = TRUE, ignore the write by T, allow T to proceed and make no change to the database.
     - If C(X) = FALSE, delay T until C(X) = TRUE or the transaction that wrote X aborts.
   - Neáu TS(T) < RT(X)
     - Rollback T (Write too late)
3. If T requests to commit, the scheduler sets C(X) = True for all the db element written by T.
4. If T requests to abort, any transaction that was waiting on the an element X that T wrote must repeat its attempt to read or write.

Example

T1	T2	T3	T4	A	B	C
420	400	425	415	tr=tw=0	tr=tw=0	tr=tw=0
			ReadA	tr=415		
ReadA				tr =420		
			WriteB		tw=415	
WriteA				tw=420		
	ReadB				T2 rollback	
		ReadB			tr=425	
	ReadA					
	WriteC					
		WriteA		tw=425		

Exampl e

T1	T2	T3	T4	A	B	C
510	550	575	500	tr=tw=0	tr=tw=0	tr=tw=0
			ReadA	tr=500		
ReadA				tr =510		
			WriteB		tw=500	
WriteA				tw=510		
	ReadB				tr=550	
		ReadB			tr=575	
	ReadA			tr=550		
	WriteC					Tw=550
		WriteA		tw=575		

Multiversion timestamps

- Is an variation of timestamping,
- Maintains old versions of database elements in addition to the current version.
- In the case a read on X causes a transaction to abort, any read request on X will be processed by reading the appropriate version of X (due to the timestamp of the transaction which requests to read X).

Multiversion timestamps

- For each db element X, there may be multiple versions to be maintained: X_1, X_2, ..., $X_{k.}$
- For each version of X, the scheduler maintains:
  - The value of the version X_i,
  - The read timestamp of X_i, $r_T(X_i)$, the largest timestamp among the timestamps of transactions that have successfully read X_i (timestamp of T now).
  - The write timestamp of X_i, $w_T(X_i)$, the timestamp of the transaction T that has created the version X_i (of X).
- When a new write $w_T(X)$ occurs, if it is legal, then a new version of db element X_t is created, t=TS(T):
  - $r_T(X_t) = w_T(X_t) = TS(T)$.
- When a read on X_i by T occurs:
  - $r_T(X_i) = \max(r_T(X_i), TS(T))$

Multiversion timestamps

1. T requests a write on X:
   - Xi is the version of X that has the highest write timestamp of all versions of X that is also less than or equal to TS(T).
     - If $r_T(X_i)$ > TS(T) then roll back the transaction T.
     - If $r_T(X_i)$ <= TS(T) then:
       - If TS(T) = $w_T(X_i)$ then the version X_i is overwritten.
       - If TS(T) > $w_T(X_i)$ then a new version Xj is created, $r_T(Xj) = w_T(Xj) = TS(T)$
2. T requests a read on X:
   - Find the version i of X that has the highest write timestamp of all versions of X that is also less than or equal to TS(T).
     - Return the value of Xi to T.
     - $r_T(X_i) = \max(r_T(X_i), TS(T))$

Example

T_1 $TS(T_1)=100$	T_2 $TS(T_2)=200$	A_0 RTS=0 WTS=0	B_0 A_1 RTS=0 WTS=0	A_2	A_3
Read(A)		WTS=0 RTS=100			
Write(A)			RTS=100 WTS=100		
	Read(A)		RTS=200 WTS=100		
	Write(A)			RTS=200 WTS=200	
	Read(B)				RTS=200 WTS=0
Read(B)					RTS=200 WTS=0
	Write(A)				RTS=200 WTS=200
Read(A)			RTS=200 WTS=100		

10. Concurrency Control by Validation

Is a type of optimistic concurrency control, where we allow transactions to access data without locks, and at the appropriate time we check that the transaction has behaved in a serializable manner.

- The validation-based scheduler will give a serializable schedule. This schedule is equivalent to the serial schedule that was based on the moment that the transactions validated successfully.
- For each transaction T, the scheduler must be told:
  - The set of database elements that T reads, called the read set, RS(T).
  - The set of database elements that T writes, called the write set, WS(T).

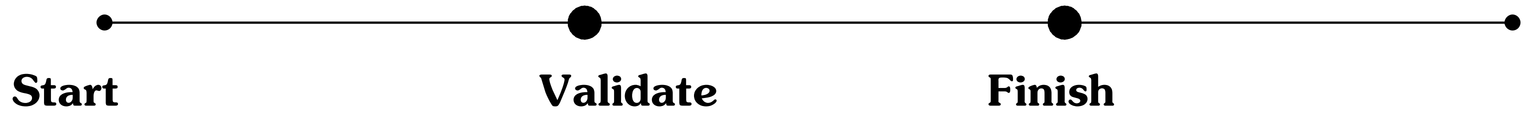
Validation

- Transactions are executed in three phases:

  1. Read phase:
     - The transaction reads from the database all the elements in its read set.
  2. Validate phase:
     - The scheduler validates the transaction by comparing its read and write sets with those of other transactions.
       - If validation fails, the transaction is rolled back.
       - Otherwise, it proceeds to the third phase.
  3. Write phase:
     - The transaction writes to the database its values for the elements in its write set.

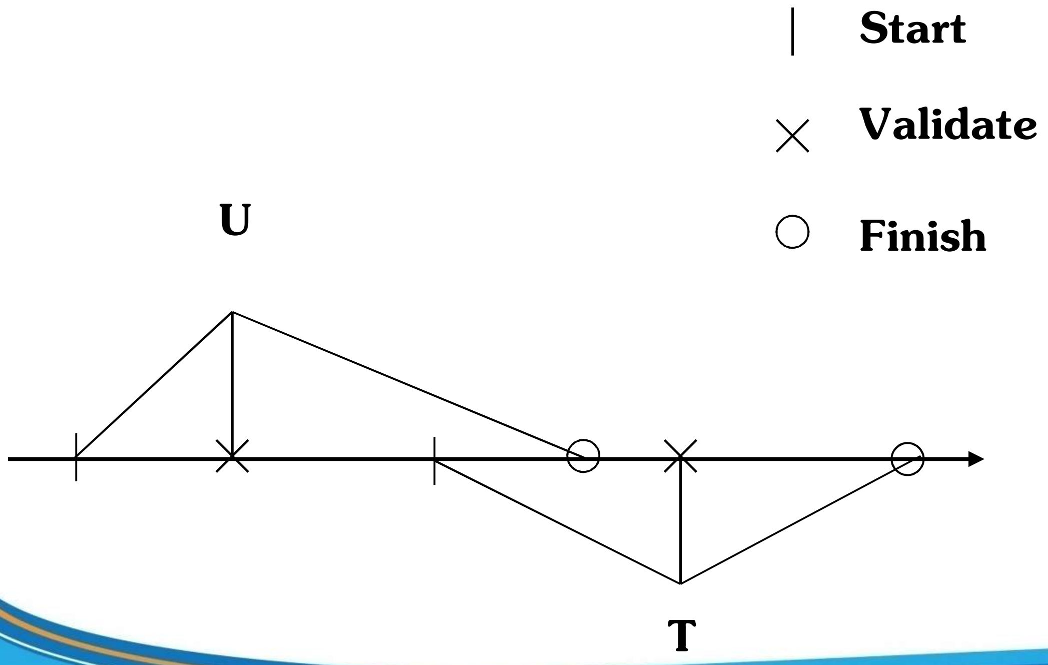
Phöông phaùp kieåm tra hôïp leä

- The scheduler maintains:
  - START(Ti) – the time at which Ti started and START – the set of transactions that have started, but not yet completed validation.
  - VAL(Ti) – the time at which Ti validated and VAL – the set of transactions that have been validated but not yet finished the writing phase (phase 3).
  - FINISH(Ti) – the time at which T finished and FIN – the set of transaction that have completed phase 3.



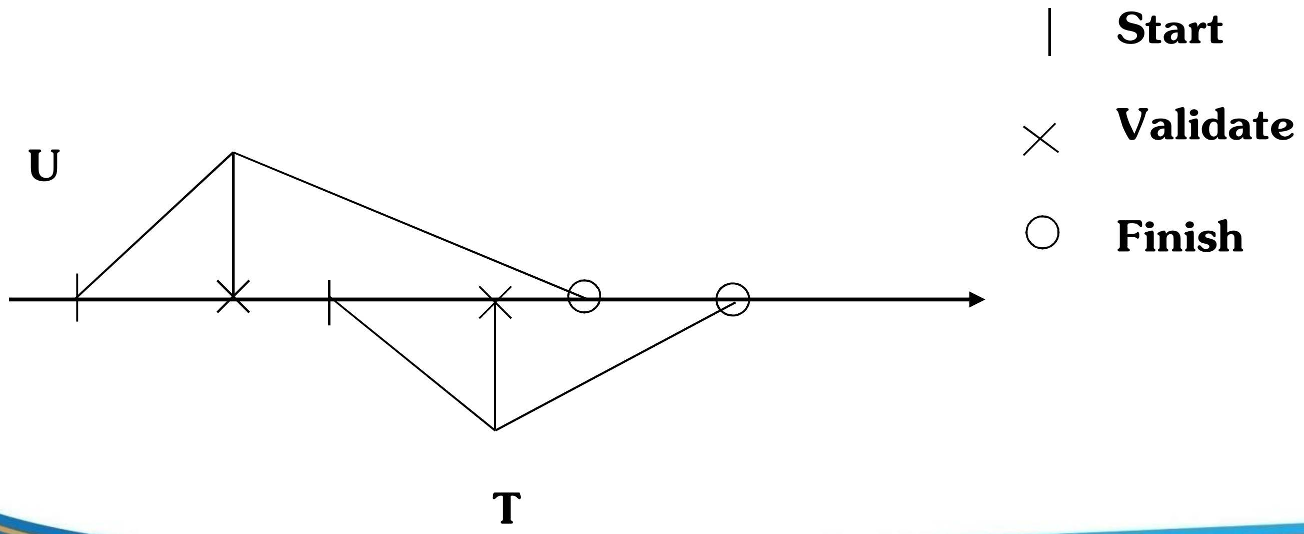
Validation Rules

- Suppose there is a transaction U such that:
  - U ∈ VAL or U ∈ FIN, that means, U has validated.
  - FIN(U) > START (T), U did not finish before T started.
  - RS(T) ∩ WS(U) = {X} ≠ ∅: U wrote X after T read X. In fact, U may not even have written X yet. We don't know whether or not T got to read U's value, we must rollback T to avoid a risk that the actions of T and U will not be consistent with the assumed serial order.

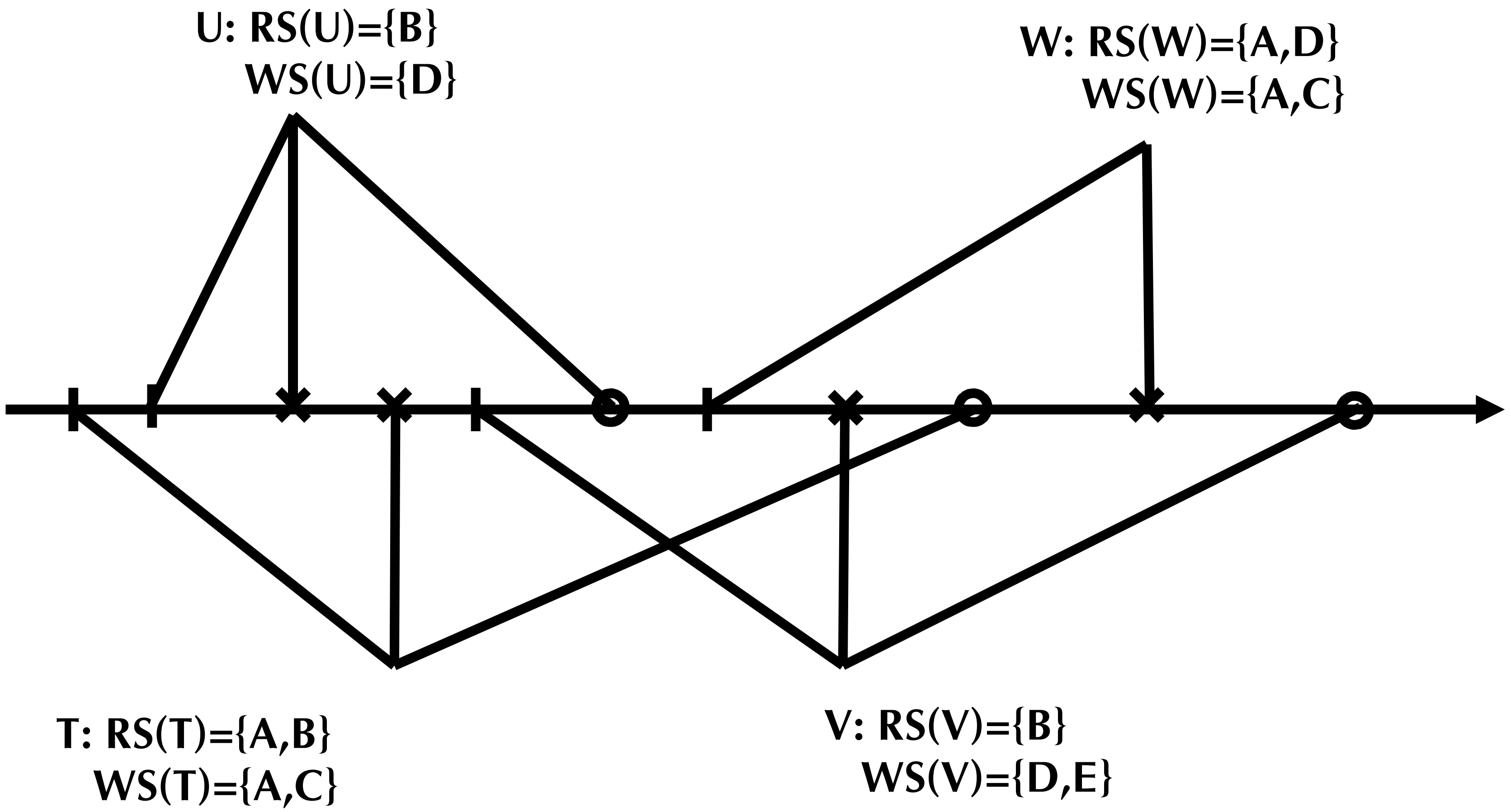


Validation Rules

- Suppose that there is a transaction U:
  - U $\in$ VAL, which means U has successfully validated.
  - FIN(U) > VAL (T), that is, U did not finish before T entered its validation phase.
  - WS(T) $\cap$ WS(U) = {X} $\neq \emptyset$: it is possible that T will write X before U does. Since we cannot be sure, we rollback T to make sure it does not violate the assumed serial order in which it follows U.

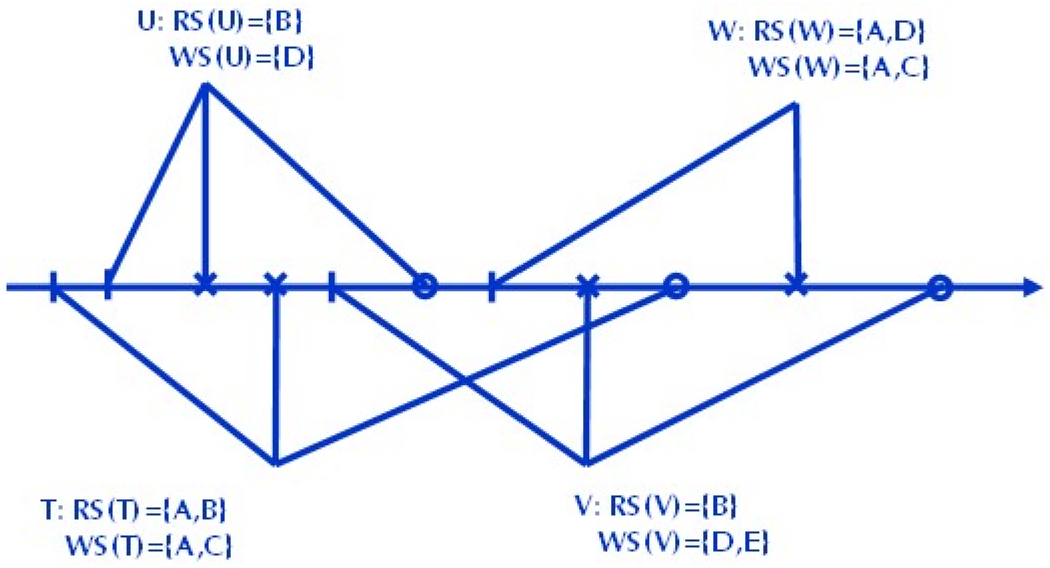


Example



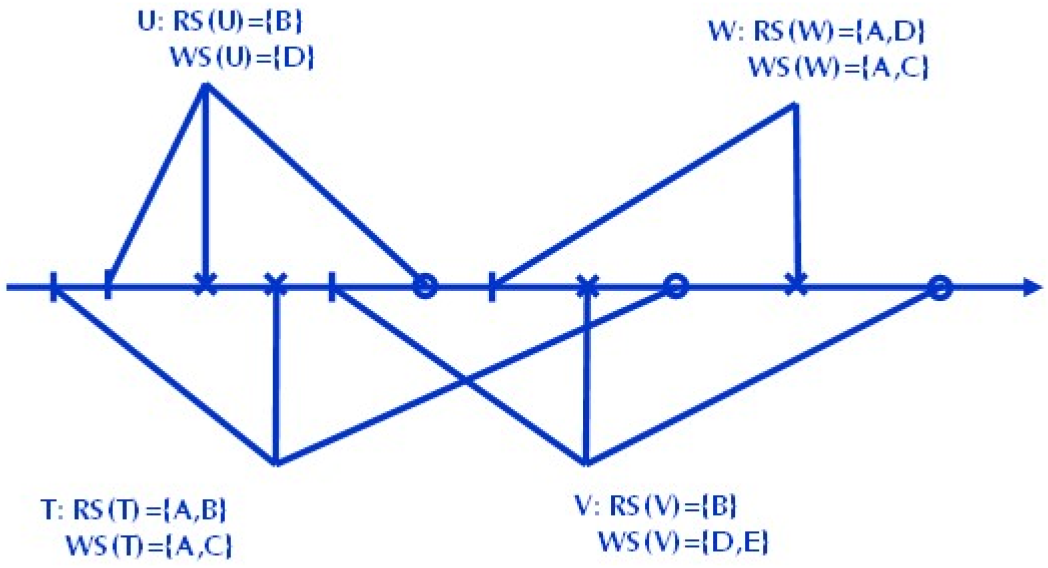
How does the validation-based scheduler work?

Example



- Validation of U:
  - When U validates there are no other validated transactions, so there is nothing to check. U validates successfully and writes a value for db element D.

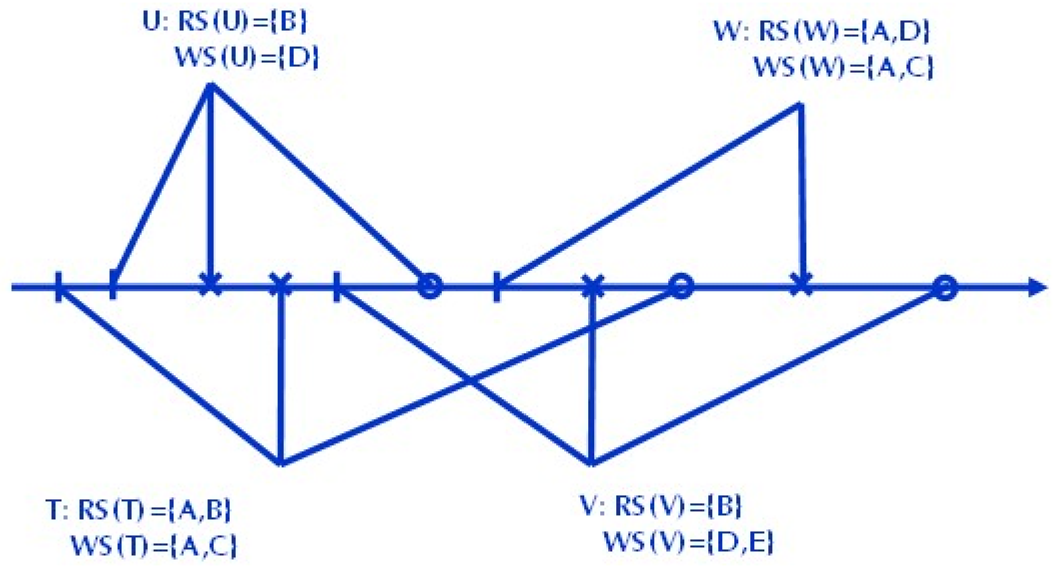
Example



- Validation of T:
  - When T validates, U is validated but not finished. We must check:
    - RS(T) ∩ WS(U) = {A, B} ∩ {D} = ∅
    - WS(T) ∩ WS(U) = {A, C} ∩ {D} = ∅

    T validates.

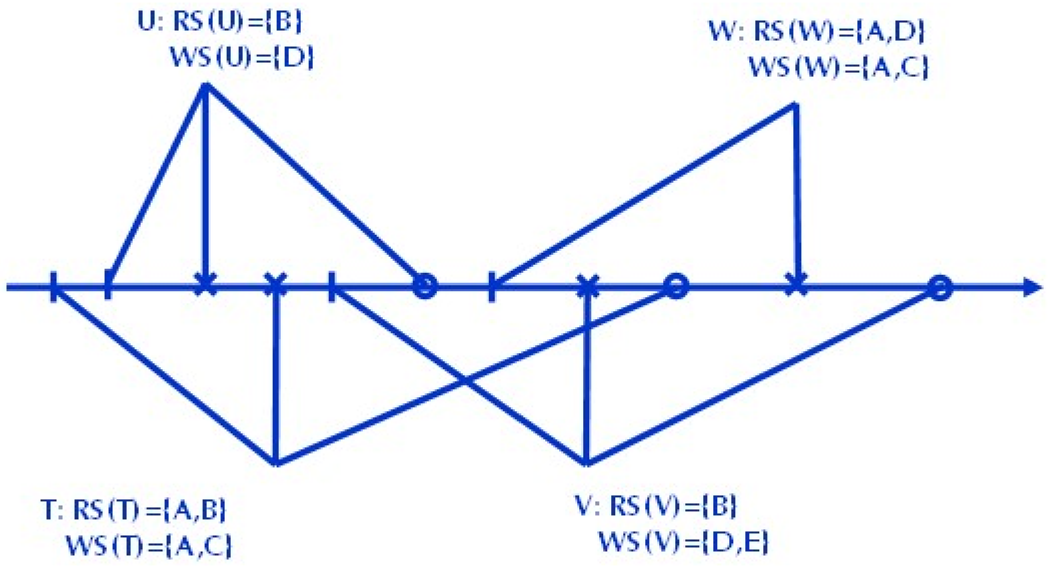
Example



- Validation of V:
  - When V validates, U is validated and finished, and T is validated but not finished. V started before U finished. We must examine:
    - RS(V) ∩ WS(U) = {B} ∩ {D} = ∅
    - RS(V) ∩ WS(T) = {B} ∩ {A, C} = ∅
    - WS(V) ∩ WS(T) = {D, E} ∩ {A, C} = ∅

V validates.

Example



- Validation of W:
  - When W validates, we find that U finished before W started, so no comparison between W and U is performed. T is finished before W validates but did not finish before W started. We examine:
    - RS(W) ∩ WS(T) = {A, D} ∩ {A, C} = {A}
    - RS(W) ∩ WS(V) = {A, D} ∩ {D, E} = {D}
    - WS(W) ∩ WS (V) = {A, C} ∩ {D, E} = ∅

  W is not validated. W is rolled back and does not write values of A or C.

Appendix: Timestamp-based CC

```
1.  T requests a Read on X
    If TS(T) >= WT(X)
        T reads X
        RT(X) := max (RT(X), TS(T))
    Else if TS(T) < WT(X)
        Rollback T (Read too late)
2.  T requests a Write on X
    If TS(T) >= RT(X) and TS(T) >= WT(X)
        T writes X
        WT(X) := TS(T)
    Else if TS(T) < RT(X) or TS(T) < WT(X)
        Rollback T (Write too late)
```

Appendix: Multi-version based CC

1. T requests a write on X:
   - Finds the version Xi of X such that wT(Xi) <=TS(T), but there is no other version Xi‘ with wT(Xi) < wT(Xi‘) <=TS(T)
     - If rT (Xi) > TS(T) then T is rolled back.
     - Else if rT (Xi) <= TS(T) then
       - If TS(T) = wT(Xi) then Xi is overwritten.
       - If TS(T) > wT(Xi) a new version Xj is created, rT (Xj) = wT(Xj) =TS(T)
2. T requests a read on X:
   - Finds the version Xi of X such that wT(Xi) <=TS(T), but there is no other version Xi‘ with wT(Xi) < wT(Xi‘) <=TS(T)
     - Return Xi.
     - rT (Xi) = max (rT (Xi), TS(T)).

End.