CSEP 544: Lecture 07

Transactions Part 2:
Concurrency Control

Announcements

- Homework 4 due next Tuesday
  - Simple for you, but reflect on TXNs

- Rest of the quarter (revised!):
  - Today: TXNs - no paper
  - 11/23: finish TXNs, Datalog - no paper
  - 11/30: Advanced Query Processing - paper
  - 12/07: Column Store, Final Review - paper

ARIES

Aries

- ARIES pieces together several techniques into a comprehensive algorithm
- Developed at IBM Almaden, by Mohan
- IBM botched the patent, so everyone uses it now
- Several variations, e.g. for distributed transactions

ARIES Recovery Manager

- A redo/undo log
- Physiological logging
  - Physical logging for REDO
  - Logical logging for UNDO
- Efficient checkpointing

ARIES Recovery Manager

Log entries:

- <START T>   -- when T begins
- Update: <T,X,u,v>
  - T updates X, *old* value=u, *new* value=v
  - In practice: *undo only* and *redo only* entries
- <COMMIT T> or <ABORT T>
- CLR’s – we’ll talk about them later.

ARIES Recovery Manager

Rule:

- If T modifies X, then <T,X,u,v> must be written to disk before OUTPUT(X)

We are free to OUTPUT early or late

LSN = Log Sequence Number

- **LSN** = identifier of a log entry
  - Log entries belonging to the same TXN are linked

- Each page contains a **pageLSN**:
  - LSN of log record for latest update to that page

ARIES Data Structures

- **Active Transactions Table**
  - Lists all active TXN's
  - For each TXN: lastLSN = its most recent update LSN
- **Dirty Page Table**
  - Lists all dirty pages
  - For each dirty page: recoveryLSN (recLSN)= first LSN that caused page to become dirty
- **Write Ahead Log**
  - LSN, prevLSN = previous LSN for same txn

$W_{T100}(P7)$
$W_{T200}(P5)$
$W_{T200}(P6)$
$W_{T100}(P5)$

ARIES Data Structures

Dirty pages

pageID	recLSN
P5	102
P6	103
P7	101

Log (WAL)

LSN	prevLSN	transID	pageID	Log entry
101	-	T100	P7	
102	-	T200	P5	
103	102	T200	P6	
104	101	T100	P5	

Active transactions

transID	lastLSN
T100	104
T200	103

Buffer Pool

P8	**P2**	. . .
	. . .	
P5 PageLSN=104	**P6** PageLSN=103	**P7** PageLSN=101

ARIES Normal Operation

T writes page P

- What do we do ?

ARIES Normal Operation

T writes page P

- What do we do ?

- Write **<T,P,u,v>** in the **Log**
- **pageLSN**=**LSN**
- **prevLSN=lastLSN**
- **lastLSN**=**LSN**
- **recLSN**=if isNull then **LSN**

ARIES Normal Operation

Buffer manager wants to OUTPUT(P)

- What do we do ?

Buffer manager wants INPUT(P)

- What do we do ?

ARIES Normal Operation

Buffer manager wants to OUTPUT(P)

- Flush log up to **pageLSN**
- Remove P from **Dirty Pages** table

Buffer manager wants INPUT(P)

- Create entry in **Dirty Pages** table
  recLSN = NULL

ARIES Normal Operation

Transaction T starts

- What do we do ?

Transaction T commits/aborts

- What do we do ?

ARIES Normal Operation

Transaction T starts

- Write **<START T>** in the log
- New entry T in Active TXN; lastLSN = null

Transaction T commits/aborts

- Write **<COMMIT T>** in the log
- Flush log up to this entry

Checkpoints

Write into the log

- Entire active transactions table
- Entire dirty pages table

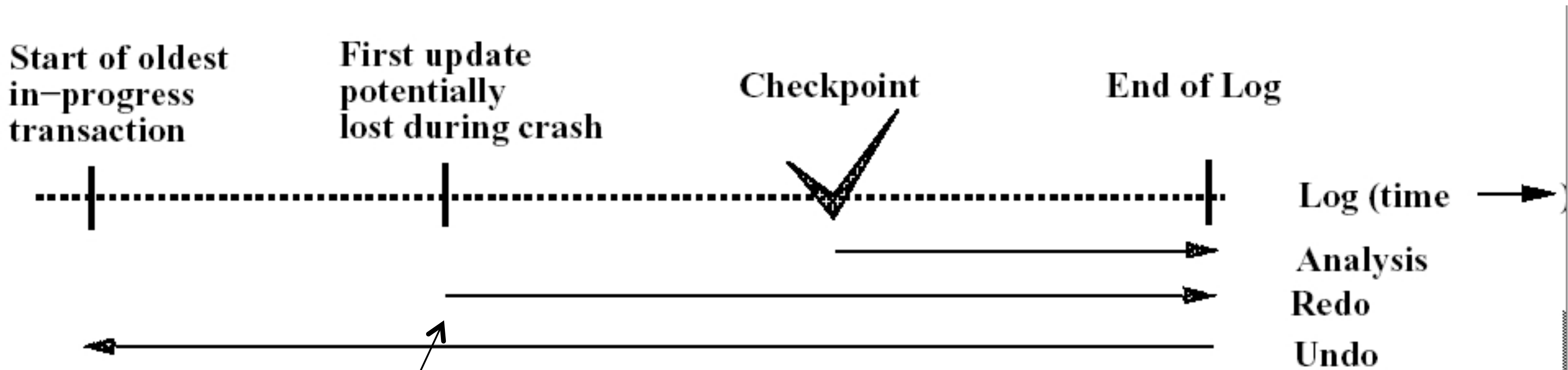
Recovery always starts by analyzing latest checkpoint

Background process periodically flushes dirty pages to disk

ARIES Recovery

1. **Analysis pass**
   - Figure out what was going on at time of crash
   - List of dirty pages and active transactions
2. **Redo pass (repeating history principle)**
   - Redo all operations, even for transactions that will not commit
   - Get back to state at the moment of the crash
3. **Undo pass**
   - Remove effects of all uncommitted transactions
   - Log changes during undo in case of another crash during undo

ARIES Method Illustration



First undo and first redo log entry might be in reverse order

[Figure 3 from Franklin97]

1. Analysis Phase

- Goal
  - Determine point in log where to start REDO
  - Determine set of dirty pages when crashed
    - Conservative estimate of dirty pages
  - Identify active transactions when crashed

- Approach
  - Rebuild active transactions table and dirty pages table
  - Reprocess the log from the checkpoint
    - Only update the two data structures
  - Compute: **firstLSN** = smallest of all recoveryLSN

1. Analysis Phase

Log Checkpoint (crash)

firstLSN= ???

Where do we start the REDO phase ?

Dirty pages

pageID	recLSN	pageID

Active txn

transID	lastLSN	transID

1. Analysis Phase

Log Checkpoint (crash)

firstLSN=min(**recLSN**)

Dirty pages

pageID	recLSN	pageID

Active txn

transID	lastLSN	transID

1. Analysis Phase

Log

Checkpoint

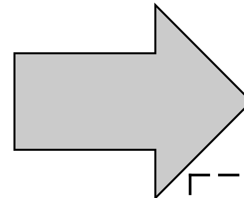
(crash)

firstLSN

Dirty pages

pageID	recLSN	pageID

Replay history



pageID	recLSN	pageID

Active txn

transID	lastLSN	transID

transID	lastLSN	transID

2. Redo Phase

Main principle: replay history

- Process Log forward, starting from **firstLSN**
- Read every log record, sequentially
- Redo actions are not recorded in the log
- Needs the Dirty Page Table

2. Redo Phase: Details

For each Log entry record LSN: **<T,P,u,v>**

- Re-do the action P=u and WRITE(P)
- But which actions can we skip, for efficiency ?

2. Redo Phase: Details

For each Log entry record LSN: **<T,P,u,v>**

- If P is not in Dirty Page then **no update**
- If recLSN > LSN, then **no update**
- Read page from disk:
  If **pageLSN** > LSN, then **no update**
- Otherwise perform update

2. Redo Phase: Details

What happens if system crashes during REDO ?

2. Redo Phase: Details

What happens if system crashes during REDO ?

We REDO again ! Each REDO operation is *idempotent*: doing it twice is the as as doing it once.

3. Undo Phase

- Cannot “unplay” history, in the same way as we “replay” history
- WHY NOT ?

3. Undo Phase

- Cannot “unplay” history, in the same way as we “replay” history
- WHY NOT ?

- Need to support ROLLBACK: selective undo, for one transaction
- Hence, *logical* undo v.s. *physical* redo

3. Undo Phase

Main principle: “logical” undo

- Start from end of Log, move backwards
- Read only affected log entries
- Undo actions *are* written in the Log as special entries: CLR (Compensating Log Records)
- CLRs are redone, but never undone

3. Undo Phase: Details

- "Loser transactions" = uncommitted transactions in Active Transactions Table
- **ToUndo** = set of lastLSN of loser transactions

3. Undo Phase: Details

While **ToUndo** not empty:

- Choose most recent (largest) LSN in **ToUndo**
- If LSN = regular record **<T,P,u,v>**:
  - Undo v
  - Write a CLR where CLR.undoNextLSN = LSN.prevLSN
- If LSN = CLR record:
  - Don't undo !
- if CLR.**undoNextLSN** not null, insert in **ToUndo** otherwise, write **<END TRANSACTION>** in log

3. Undo Phase: Details

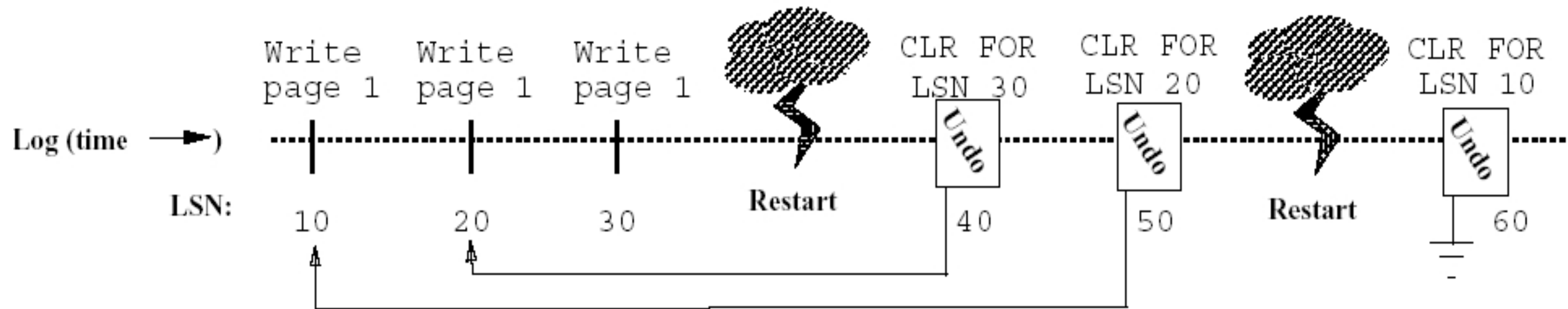


Figure 4: The Use of CLRs for UNDO

[Figure 4 from Franklin97]

3. Undo Phase: Details

What happens if system crashes during UNDO ?

3. Undo Phase: Details

What happens if system crashes during UNDO ?

We do not UNDO again ! Instead, each CLR is a REDO record: we simply redo the undo

Physical v.s. Logical Loging

Why are redo records *physical* ?

Why are undo records *logical* ?

Physical v.s. Logical Loging

Why are redo records *physical* ?

- Simplicity: replaying history is easy, and idempotent

Why are undo records *logical* ?

- Required for transaction rollback: this not "undoing history", but selective undo

Concurrency Control

Recap ACID:

- Atomicity – recovery
- Consistency
- Isolation – concurrency control
- Durability

Reading Material

Main textbook (Ramakrishnan and Gehrke):

- Chapters 16, 17, 18

More background material: Garcia-Molina, Ullman, Widom:

- Chapters 17.2, 17.3, 17.4
- Chapters 18.1, 18.2, 18.3, 18.8, 18.9

Concurrency Control

- Multiple concurrent transactions T_1, T_2, …
- They read/write common elements A_1, A_2, …
- How can we prevent unwanted interference ?

The SCHEDULER is responsible for that

Schedules

A *schedule* is a sequence of interleaved actions from all transactions

Example

A and B are elements in the database
t and s are variables in tx source code

T1	T2
READ(A, t)	READ(A, s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A,s)
READ(B, t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

A 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*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

Serializable Schedule

A schedule is ***serializable*** if it is equivalent to a serial schedule

A Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(B,s)
	s := s*2
	WRITE(B,s)

This is a serializable schedule.
This is NOT a serial schedule

A Non-Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Why is it non-serializable?

Serializable Schedules

- The role of the scheduler is to ensure that the schedule is serializable

Q: Why not run only serial schedules ?
I.e. run one transaction after the other ?

Serializable Schedules

- The role of the scheduler is to ensure that the schedule is serializable

Q: Why not run only serial schedules ?
I.e. run one transaction after the other ?

A: Because of very poor throughput due to disk latency.

Lesson: main memory databases *may* do serial schedules only

Still Serializable, but…

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s + 200
	WRITE(A,s)
	READ(B,s)
	s := s + 200
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Schedule is serializable because t=t+100 and s=s+200 commute

…we don't expect the scheduler to schedule this

Ignoring Details

- Assume worst case updates:
  - We never commute actions done by transactions
- As a consequence, we only care about reads and writes
  - Transaction = sequence of R(A)'s and W(A)'s

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

Conflicts

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW

Conflict Serializability

Conflicts:

Two actions by same transaction T_i: $r_i(X); w_i(Y)$

Two writes by T_i, T_j to same element $w_i(X); w_j(X)$

Read/write by T_i, T_j to same element $w_i(X); r_j(X)$

$r_i(X); w_j(X)$

Conflict Serializability

Definition A schedule is *conflict serializable* if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions

- Every conflict-serializable schedule is serializable
- The converse is not true in general

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B)$

....

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Testing for Conflict-Serializability

Precedence graph:

- A node for each transaction T_i,
- An edge from T_i to T_j whenever an action in T_i conflicts with, and comes before an action in T_j

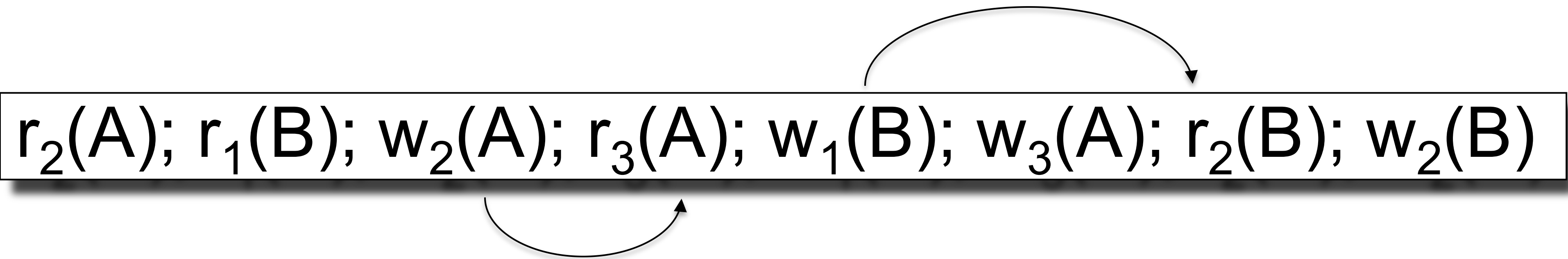
- The schedule is serializable iff the precedence graph is acyclic

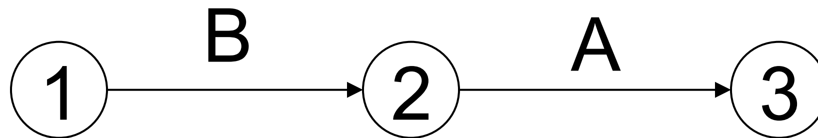
Example 1

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

1 2 3

Example 1





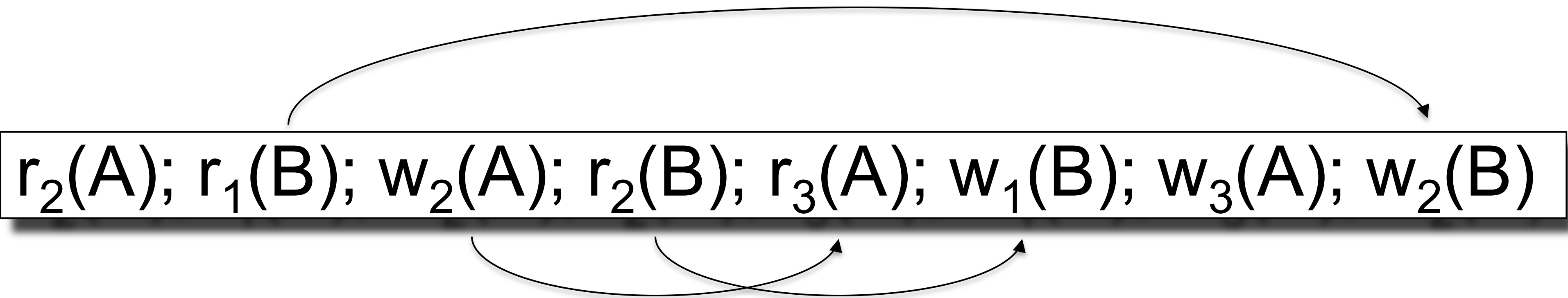
This schedule is conflict-serializable

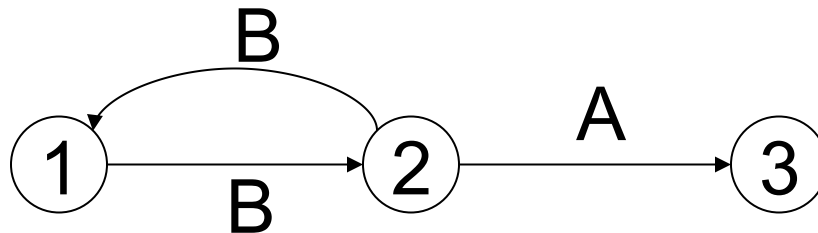
Example 2

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

1 2 3

Example 2





This schedule is NOT conflict-serializable

View Equivalence

- A serializable schedule need not be conflict serializable, even under the "worst case update" assumption

$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

Is this schedule conflict-serializable ?

View Equivalence

- A serializable schedule need not be conflict serializable, even under the "worst case update" assumption

$$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$$

Is this schedule conflict-serializable ?

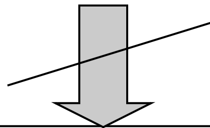
No…

View Equivalence

- A serializable schedule need not be conflict serializable, even under the "worst case update" assumption

$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

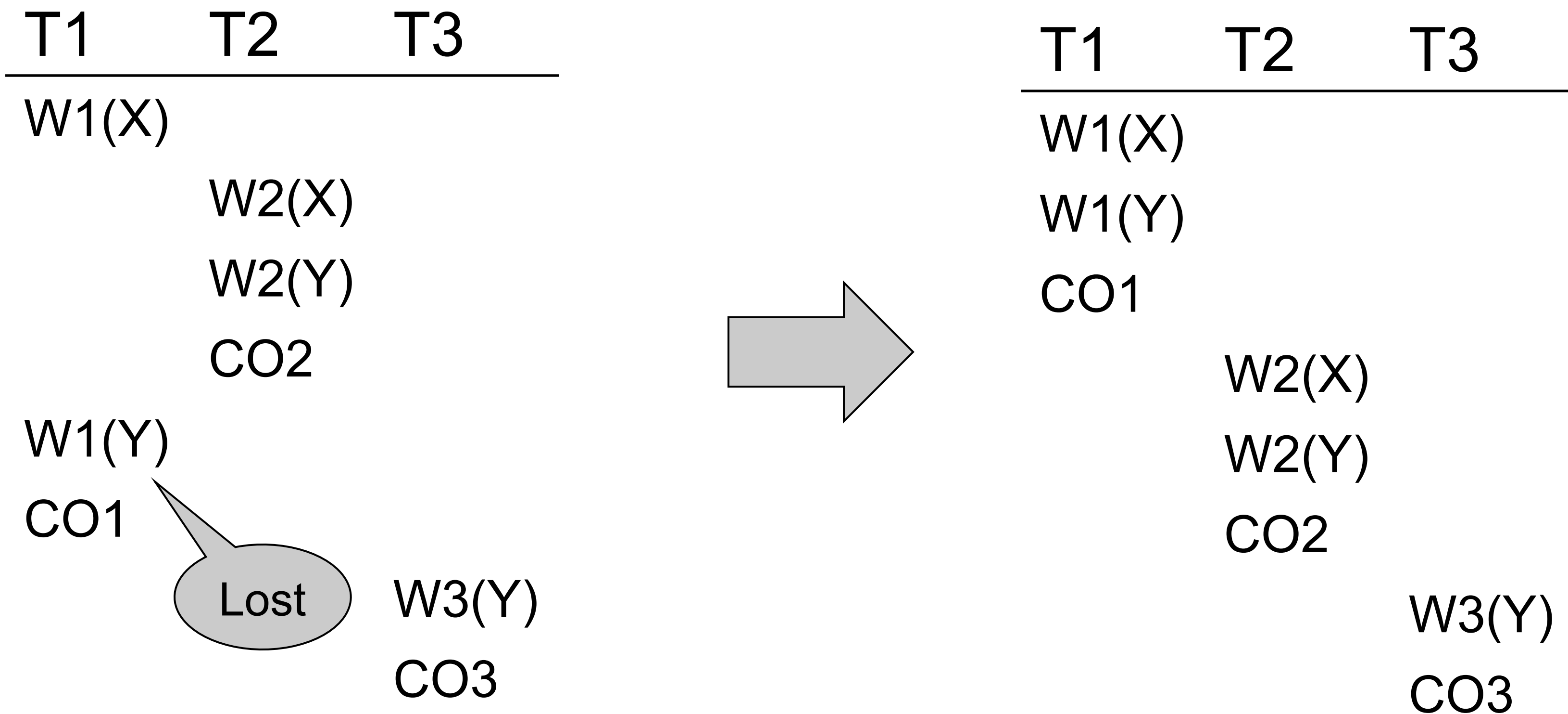
Lost write



$w_1(X); w_1(Y); w_2(X); w_2(Y); w_3(Y);$

Equivalent, but not conflict-equivalent

View Equivalence



Serializable, but not conflict serializable

View Equivalence

Two schedules S, S’ are *view equivalent* if:

- If T reads an initial value of A in S,
  then T reads the initial value of A in S’
- If T reads a value of A written by T’ in S,
  then T reads a value of A written by T’ in S’
- If T writes the final value of A in S,
  then T writes the final value of A in S’

View-Serializability

A schedule is *view serializable* if it is view equivalent to a serial schedule

Remark:

- If a schedule is *conflict serializable*, then it is also *view serializable*
- But not vice versa

Schedules with Aborted Transactions

- When a transaction aborts, the recovery manager undoes its updates
- But some of its updates may have affected other transactions !

Schedules with Aborted Transactions

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

What's wrong?

Schedules with Aborted Transactions

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

What's wrong?

Cannot abort T1 because cannot undo T2

Recoverable Schedules

A schedule is *recoverable* if:

- ~~It is conflict-serializable, and~~
- Whenever a transaction T commits, all transactions who have written elements read by T have already committed

Recoverable Schedules

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
?	

Nonrecoverable

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

Recoverable

Recoverable Schedules

T1	T2	T3	T4
R(A)			
W(A)			
	R(A)		
	W(A)		
	R(B)		
	W(B)		
		R(B)	
		W(B)	
		R(C)	
		W(C)	
			R(C)
			W(C)
			R(D)
			W(D)
Abort			

How do we recover ?

Cascading Aborts

- If a transaction T aborts, then we need to abort any other transaction T' that has read an element written by T
- A schedule *avoids cascading aborts* if whenever a transaction reads an element, the transaction that has last written it has already committed.

Avoiding Cascading Aborts

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
. . .	
	. . .

With cascading aborts

T1	T2
R(A)	
W(A)	
Commit	
	R(A)
	W(A)
	R(B)
	W(B)
	. . .

Without cascading aborts

Review of Schedules

Serializability

- Serial
- Serializable
- Conflict serializable
- View serializable

Recoverability

- Recoverable
- Avoids cascading deletes

Scheduler

- The scheduler:
- Module that schedules the transaction's actions, ensuring serializability

- Two main approaches
- Pessimistic: locks
- Optimistic: time stamps, MV, validation

Pessimistic Scheduler

Simple idea:

- Each element has a unique lock
- Each transaction must first acquire the lock before reading/writing that element
- If the lock is taken by another transaction, then wait
- The transaction must release the lock(s)

Notation

$l_i(A)$ = transaction T_i acquires lock for element A

$u_i(A)$ = transaction T_i releases lock for element A

A Non-Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Example

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

Scheduler has ensured a conflict-serializable schedule

But…

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

Locks did not enforce conflict-serializability !!! What's wrong ?

Two Phase Locking (2PL)

The 2PL rule:

- In every transaction, all lock requests must preceed all unlock requests

- This ensures conflict serializability ! (will prove this shortly)

Example: 2PL transactions

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

Now it is conflict-serializable

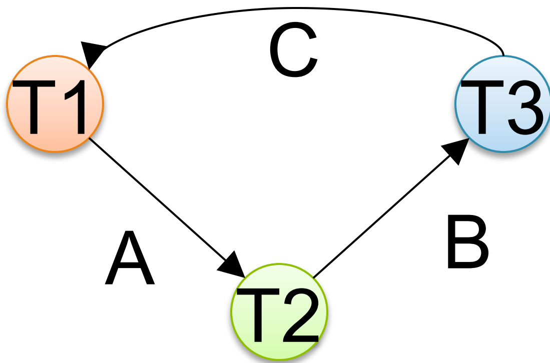
Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

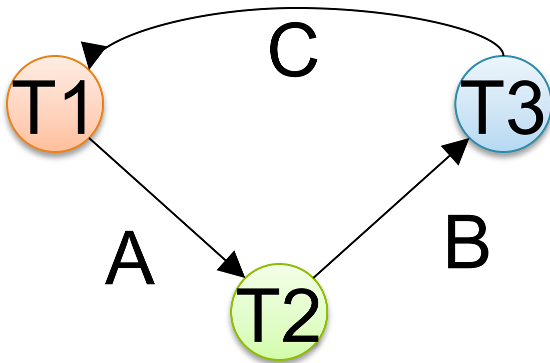
Proof. Suppose not: then there exists a cycle in the precedence graph.



Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.

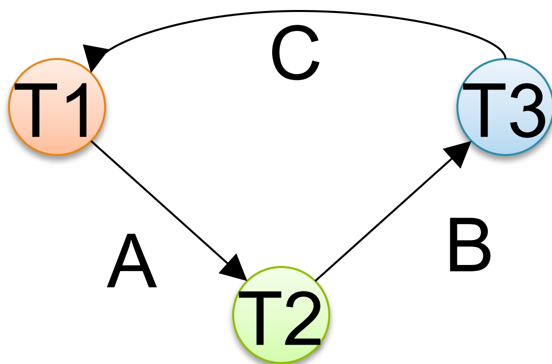


Then there is the following **temporal** cycle in the schedule:

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



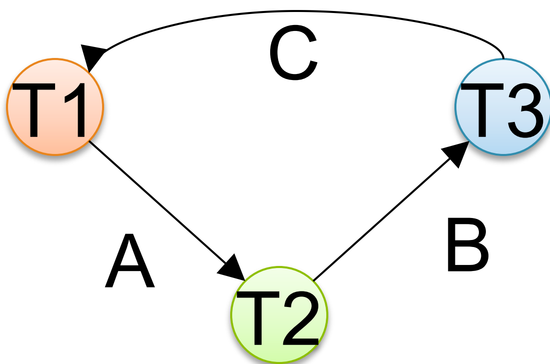
Then there is the following **temporal** cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$ why?

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



Then there is the following **temporal** cycle in the schedule:

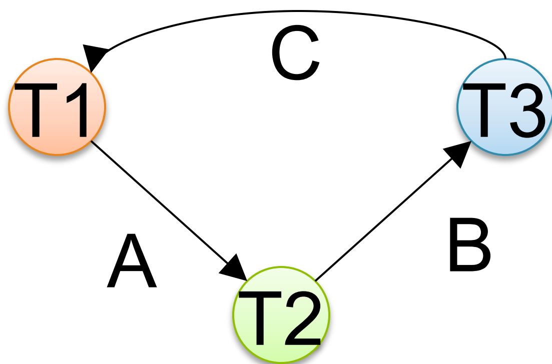
$U_1(A) \rightarrow L_2(A)$

$L_2(A) \rightarrow U_2(B)$ why?

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



Then there is the following **temporal** cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$

$L_2(A) \rightarrow U_2(B)$

$U_2(B) \rightarrow L_3(B)$

$L_3(B) \rightarrow U_3(C)$

$U_3(C) \rightarrow L_1(C)$

$L_1(C) \rightarrow U_1(A)$

Contradiction

A New Problem: Non-recoverable Schedule

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

Strict 2PL

- Strict 2PL: All locks held by a transaction are released when the transaction is completed; release happens at the time of COMMIT or ROLLBACK
- Schedule is recoverable
- Schedule avoids cascading aborts
- Schedule is strict: read book

Strict 2PL

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

Summary of Strict 2PL

- Ensures serializability, recoverability, and avoids cascading aborts

- Issues: implementation, lock modes, granularity, deadlocks, performance

The Locking Scheduler

Task 1: -- act on behalf of the transaction

Add lock/unlock requests to transactions

- Examine all READ(A) or WRITE(A) actions
- Add appropriate lock requests
- On COMMIT/ROLLBACK release all locks
- Ensures Strict 2PL !

The Locking Scheduler

Task 2: -- act on behalf of the system
Execute the locks accordingly

- Lock table: a big, critical data structure in a DBMS !
- When a lock is requested, check the lock table
  - Grant, or add the transaction to the element's wait list
- When a lock is released, re-activate a transaction from its wait list
- When a transaction aborts, release all its locks
- Check for deadlocks occasionally

Lock Modes

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)

Lock compatibility matrix:

	None	S	X
None	OK	OK	OK
S	OK	OK	Conflict
X	OK	Conflict	Conflict

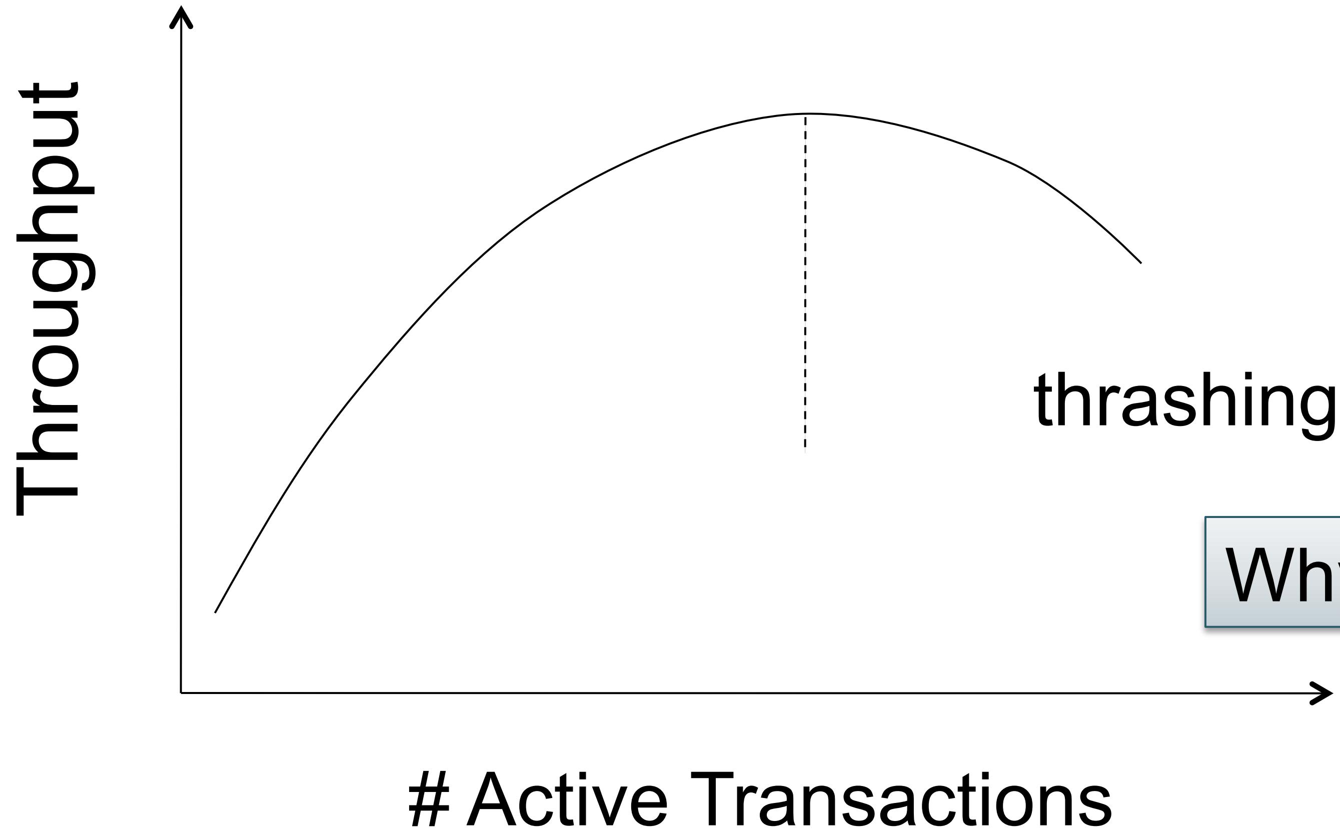
Lock Granularity

- Fine granularity locking (e.g., tuples)
  - High concurrency
  - High overhead in managing locks
- Coarse grain locking (e.g., tables, predicate locks)
  - Many false conflicts
  - Less overhead in managing locks
- Alternative techniques
  - Hierarchical locking (and intentional locks) [commercial DBMSs]
  - Lock escalation

Deadlocks

- Cycle in the wait-for graph:
  - T1 waits for T2
  - T2 waits for T3
  - T3 waits for T1
- Deadlock detection
  - Timeouts
  - Wait-for graph
- Deadlock avoidance
  - Acquire locks in pre-defined order
  - Acquire all locks at once before starting

Lock Performance



The Tree Protocol

- An alternative to 2PL, for tree structures
- E.g. B-trees (the indexes of choice in databases)
- Because
  - Indexes are hot spots!
  - 2PL would lead to great lock contention

The Tree Protocol

Rules:

- The first lock may be any node of the tree
- Subsequently, a lock on a node A may only be acquired if the transaction holds a lock on its parent B
- Nodes can be unlocked in any order (no 2PL necessary)
- "Crabbing"
  - First lock parent then lock child
  - Keep parent locked only if may need to update it
  - Release lock on parent if child is not full

- The tree protocol is NOT 2PL, yet ensures conflict-serializability !

Phantom Problem

- So far we have assumed the database to be a *static* collection of elements (=tuples)
- If tuples are inserted/deleted then the *phantom problem* appears

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color=‘blue’	
	INSERT INTO Product(name, color) VALUES (‘gizmo’,’blue’)
SELECT * FROM Product WHERE color=‘blue’	

Is this schedule serializable ?

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color=‘blue’	
	INSERT INTO Product(name, color) VALUES (‘gizmo’,’blue’)
SELECT * FROM Product WHERE color=‘blue’	

Suppose there are two blue products, X1, X2:

R1(X1),R1(X2),W2(X3),R1(X1),R1(X2),R1(X3)

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color=‘blue’	
	INSERT INTO Product(name, color) VALUES (‘gizmo’,’blue’)
SELECT * FROM Product WHERE color=‘blue’	

Suppose there are two blue products, X1, X2:

R1(X1),R1(X2),W2(X3),R1(X1),R1(X2),R1(X3)

This is conflict serializable ! What’s wrong ??

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color=‘blue’	
	INSERT INTO Product(name, color) VALUES (‘gizmo’,’blue’)
SELECT * FROM Product WHERE color=‘blue’	

Suppose there are two blue products, X1, X2:

R1(X1),R1(X2),W2(X3),R1(X1),R1(X2),R1(X3)

Not serializable due to ***phantoms***

Phantom Problem

- A “phantom” is a tuple that is invisible during part of a transaction execution but not invisible during the entire execution

- In our example:
  - T1: reads list of products
  - T2: inserts a new product
  - T1: re-reads: a new product appears !

Phantom Problem

- In a ***static*** database:
  - Conflict serializability implies serializability
- In a ***dynamic*** database, this may fail due to phantoms
- Strict 2PL guarantees conflict serializability, but not serializability

Dealing With Phantoms

- Lock the entire table, or
- Lock the index entry for ‘blue’
  - If index is available
- Or use predicate locks
  - A lock on an arbitrary predicate

Dealing with phantoms is expensive !

Isolation Levels in SQL

1. “Dirty reads”
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. “Committed reads”
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. “Repeatable reads”
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

1. Isolation Level: Dirty Reads

- "Long duration" WRITE locks
  - Strict 2PL
- No READ locks
  - Read-only transactions are never delayed

Possible pbs: dirty and inconsistent reads

2. Isolation Level: Read Committed

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

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

3. Isolation Level: Repeatable Read

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

This is not serializable yet !!!

Why ?

4. Isolation Level Serializable

- “Long duration” WRITE locks
  - Strict 2PL
- “Long duration” READ locks
  - Strict 2PL
- Deals with phantoms too

READ-ONLY Transactions

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

          DELETE  FROM Product
                  WHERE price <=0.99
          COMMIT

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

          SELECT count(*)
          FROM SmallProduct
          COMMIT
```

Optimistic Concurrency Control Mechanisms

- Pessimistic:
  - Locks

- Optimistic
  - Timestamp based: basic, multiversion
  - Validation
  - Snapshot isolation: a variant of both

Timestamps

- Each transaction receives a unique timestamp TS(T)

Could be:

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

Timestamps

Main invariant:

The timestamp order defines
the serialization order of the transaction

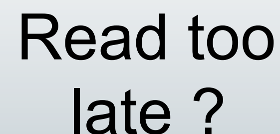
Will generate a schedule that is view-equivalent to a serial schedule, and recoverable

Main Idea

- For any two conflicting actions, ensure that their order is the serialized order:

Check WT, RW, WW conflicts

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



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

Timestamps

With each element X, associate

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

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

Simplified Timestamp-based Scheduling

Only for transactions that do not abort

Otherwise, may result in non-recoverable schedule

Transaction wants to read element X

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

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

Transaction wants to write element X

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

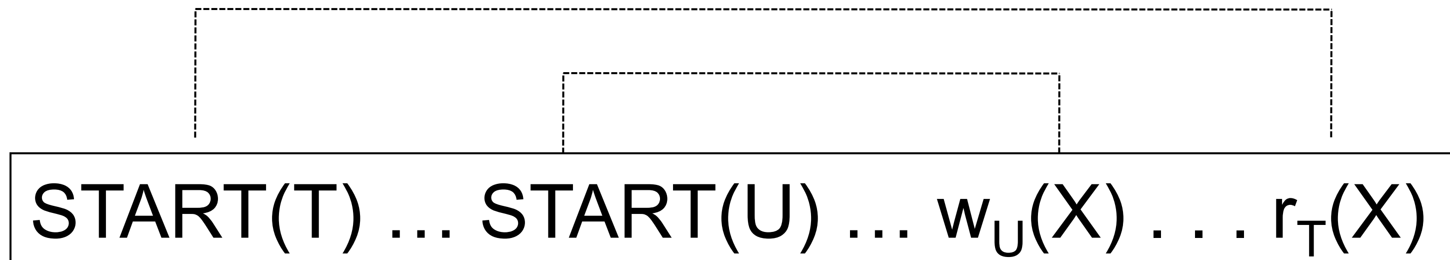
Else if WT(X) > TS(T) ignore write & continue (Thomas Write Rule)

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

Details

Read too late:

- T wants to read X, and WT(X) > TS(T)

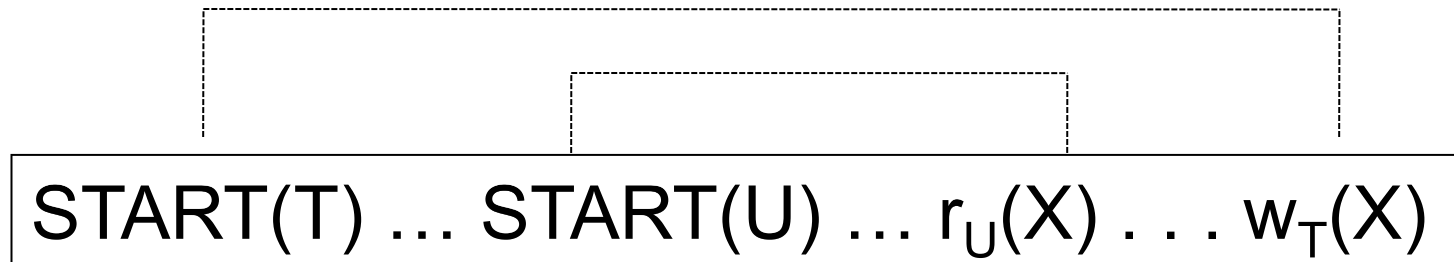


Need to rollback T !

Details

Write too late:

- T wants to write X, and RT(X) > TS(T)

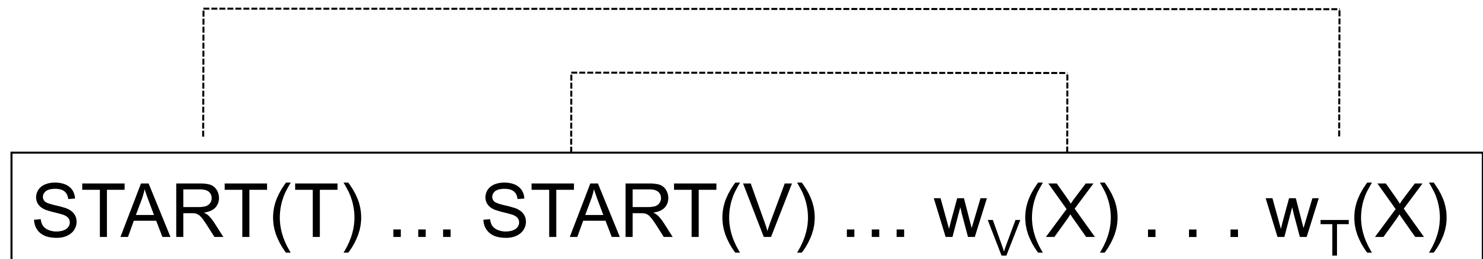


Need to rollback T !

Details

Write too late, but we can still handle it:

- T wants to write X, and
  RT(X) ≤ TS(T) but WT(X) > TS(T)



Don't write X at all !
(Thomas' rule)

View-Serializability

- By using Thomas' rule we do not obtain a conflict-serializable schedule
- But we obtain a view-serializable schedule

Ensuring Recoverable Schedules

- Recall the definition: if a transaction reads an element, then the transaction that wrote it must have already committed
- Use the commit bit C(X) to keep track if the transaction that last wrote X has committed

Ensuring Recoverable Schedules

Read dirty data:

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

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

Ensuring Recoverable Schedules

Thomas' rule needs to be revised:

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

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

Timestamp-based Scheduling

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

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

Summary of Timestamp-based Scheduling

- View-serializable
- Recoverable
  - Even avoids cascading aborts
- Does NOT handle phantoms
  - These need to be handled separately, e.g. predicate locks

Multiversion Timestamp

- When transaction T requests r(X) but WT(X) > TS(T), then T must rollback
- Idea: keep multiple versions of X: $X_t, X_{t-1}, X_{t-2}, \ldots$

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

- Let T read an older version, with appropriate timestamp

Details

- When $w_T(X)$ occurs,
  create a new version, denoted X_t where t = TS(T)

- When $r_T(X)$ occurs,
  find most recent version X_t such that t < TS(T)
  Notes:
  – $WT(X_t)$ = t and it never changes
  – $RT(X_t)$ must still be maintained to check legality of writes

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

Example (in class)

X_3 X_9 X_{12} X_{18}

R6(X) -- what happens?

W14(X) – what happens?

R15(X) – what happens?

W5(X) – what happens?

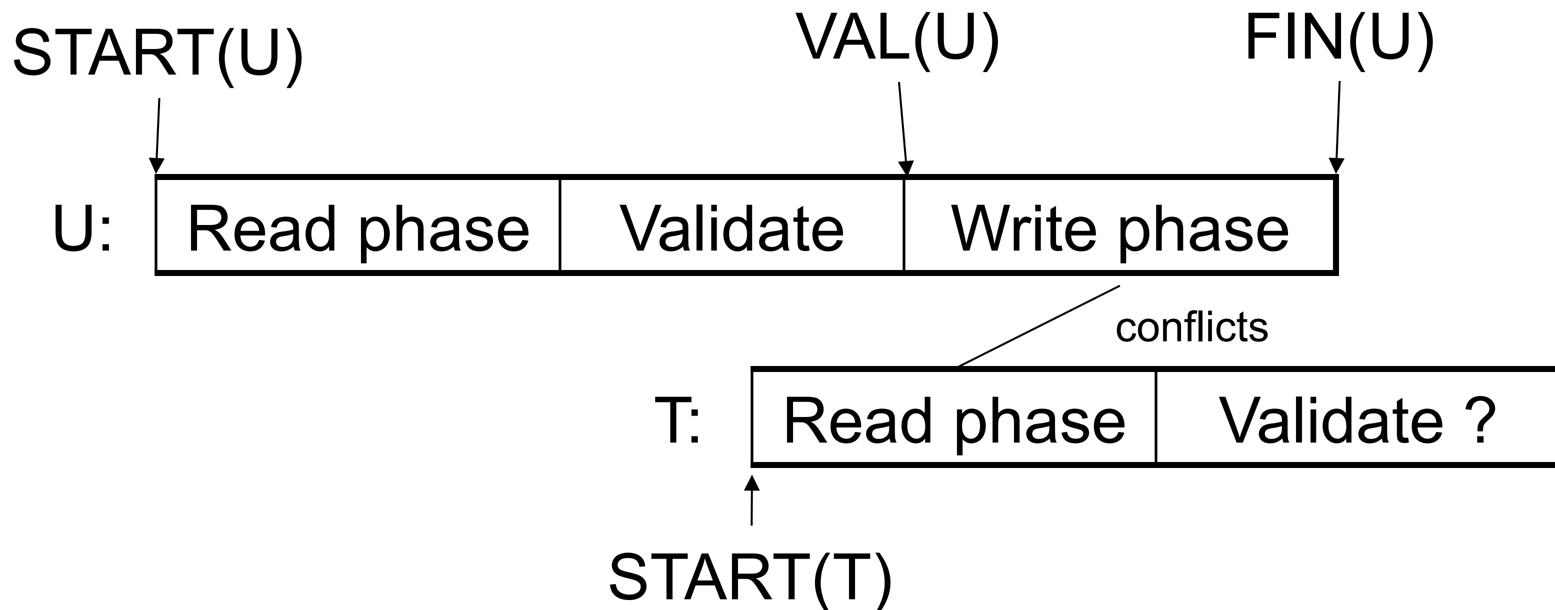
When can we delete X_3?

Concurrency Control by Validation

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

Main invariant: the serialization order is VAL(T)

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

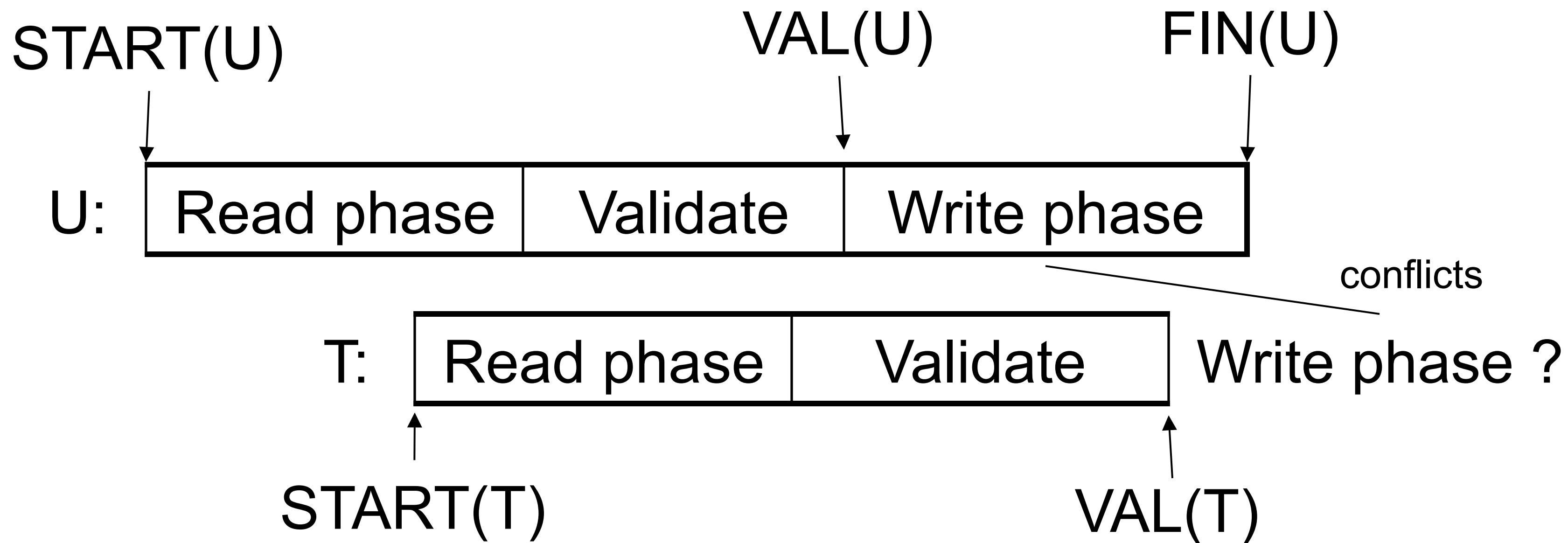


IF RS(T) ∩ WS(U) and FIN(U) > START(T)

(U has validated and U has not finished before T begun)

Then ROLLBACK(T)

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



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

Snapshot Isolation

- Another optimistic concurrency control method
- Very efficient, and very popular
  - Oracle, Postgres, SQL Server 2005

WARNING: Not serializable, yet ORACLE uses it even for SERIALIZABLE transactions !

Snapshot Isolation Rules

- Each transactions receives a timestamp TS(T)
- Tnx sees the snapshot at time TS(T) of database
- When T commits, updated pages written to disk
- Write/write conflicts are resolved by the "**first committer wins**" rule

Snapshot Isolation (Details)

- Multiversion concurrency control:
  - Versions of X: X_{t1}, X_{t2}, X_{t3}, . . .
- When T reads X, return $X_{TS(T)}$.
- When T writes X (to avoid lost update):
- If latest version of X is TS(T) then proceed
- If C(X) = true then abort
- If C(X) = false then wait

What Works and What Not

- No dirty reads (Why ?)
- No unconsistent reads (Why ?)
- No lost updates (“first committer wins”)

- Moreover: no reads are ever delayed

- However: read-write conflicts not caught !

Write Skew

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

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

In our notation:

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

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

Write Skews Can Be Serious

- ACIDland had two viceroys, Delta and Rho
- Budget had two registers: ta**X**es, and spend**Y**ng
- They had HIGH taxes and LOW spending…

```
Delta:
  READ(X);
  if X= 'HIGH'
      then { Y= 'HIGH';
             WRITE(Y) }
  COMMIT
```

```
Rho:
  READ(Y);
  if Y= 'LOW'
      then {X= 'LOW';
             WRITE(X) }
  COMMIT
```

… and they ran a deficit ever since.

Tradeoffs

- Pessimistic Concurrency Control (Locks):
  - Great when there are many conflicts
  - Poor when there are few conflicts
- Optimistic Concurrency Control (Timestamps):
  - Poor when there are many conflicts (rollbacks)
  - Great when there are few conflicts

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

Commercial Systems

- DB2: Strict 2PL
- SQL Server:
  - Strict 2PL for standard 4 levels of isolation
  - Multiversion concurrency control for snapshot isolation
- PostgreSQL, Oracle
  - Snapshot isolation even for SERIALIZABLE
  - Postgres introduced novel, serializable scheduler in postgres 9.1