Lecture 5
Transactions

Wednesday

October 27th, 2010

Announcement

- HW3: due next week
  - “Each customer has exactly one rental plan”
  - A many-one relationship: NO NEW TABLE !
  - Postgres available on cubist
- HW4: due in two weeks
  - Problems from both textbooks
  - Read corresponding chapters + slides

Where We Are (1/2)

Transactions:

- Recovery:
  - Have discussed simple UNDO/REDO recovery last lecture
- Concurrency control:
  - Have discussed serializability last lecture
  - Will discuss lock-based scheduler today

Where We Are (2/2)

Also today and next time:

- Weak Isolation Levels in SQL
- Advanced recovery
  - ARIES
- Advanced concurrency control
  - Timestamp based algorithms, including snapshot isolation

Review Questions

Query Answering Using Views, by Halevy

- Q1: define the problem
- Q2: how is this used for physical data independence ?
- Q3: what is *data integration* and what is its connection to query answering using views ?

Review Questions

- What is a *schedule* ?
- What is a *serializable* schedule ?
- What is a *conflict* ?
- What is a *conflict-serializable* schedule ?
- What is a *view-serializable* schedule ?
- What is a *recoverable* schedule ?
- When does a schedule avoid *cascading aborts* ?

Scheduler

- The scheduler is the module that schedules the transaction's actions, ensuring serializability
- Two main approaches
  - Pessimistic scheduler: uses locks
  - Optimistic scheduler: time stamps, validation

Locking 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

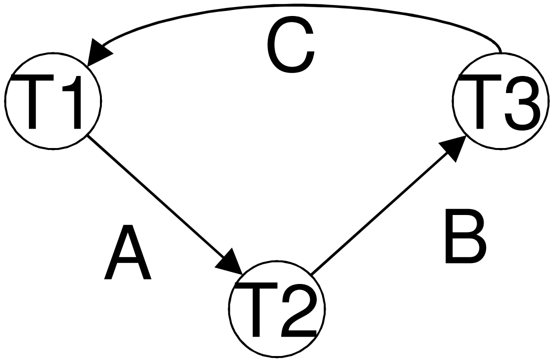
T1	T2
L_1(A); L_1(B); READ(A, t)	
t := t+100	
WRITE(A, t); U_1(A)	
	L_2(A); READ(A,s)
	s := s*2
	WRITE(A,s);
	L_2(B); **DENIED…**
READ(B, t)	
t := t+100	
WRITE(B,t); U_1(B);	
	…**GRANTED;** READ(B,s)
	s := s*2
	WRITE(B,s); U_2(A); U_2(B);

Now it is conflict-serializable

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

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); $U_2(A)$; $U_2(B)$;
Abort	**Commit**

What about Aborts?

- 2PL enforces conflict-serializable schedules
- But does not enforce recoverable schedules

Strict 2PL

- Strict 2PL: All locks held by a transaction are released when the transaction is completed
- Schedule is recoverable
  - Transactions commit only after all transactions whose changes they read also commit
- Schedule avoids cascading aborts
  - Transactions read only after the txn that wrote that element committed
- Schedule is strict: read book

Lock Modes

Standard:

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

Lots of fancy locks:

- U = update lock
  - Initially like S
  - Later may be upgraded to X
- I = increment lock (for A := A + something)
  - Increment operations commute

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

- Trasaction T_1 waits for a lock held by T_2;
- But T_2 waits for a lock held by T_3;
- While T_3 waits for . . . .
- . . .
- . . .and T_{73} waits for a lock held by T_1 !!

Deadlocks

- **Deadlock avoidance**
  - Acquire locks in pre-defined order
  - Acquire all locks at once before starting
- **Deadlock detection**
  - Timeouts
  - Wait-for graph (this is what commercial systems use)

The Locking Scheduler

Task 1:
Add lock/unlock requests to transactions

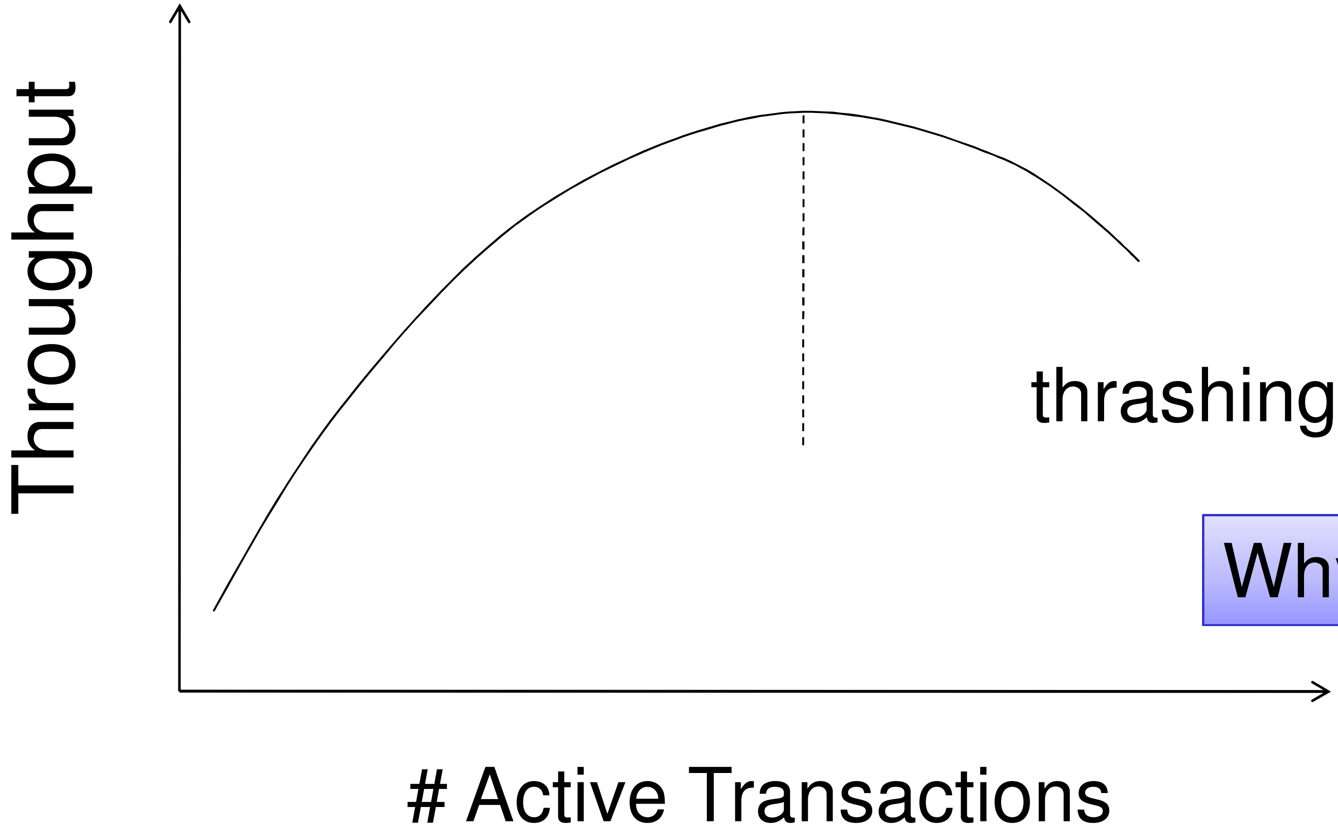
- Examine all READ(A) or WRITE(A) actions
- Add appropriate lock requests
- Ensure Strict 2PL !

The Locking Scheduler

Task 2:
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 Performance



Why ?

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)

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 all of it.

- 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 !

Degrees of Isolation

- Isolation level “serializable” (i.e. ACID)
  - Golden standard
  - Requires strict 2PL and predicate locking
  - But often too inefficient
  - Imagine there are few update operations and many long read operations

- Weaker isolation levels
  - Sacrifice correctness for efficiency
  - Often used in practice (often **default**)
  - Sometimes are hard to understand

Degrees of Isolation in SQL

- **Four levels of isolation**
  - All levels use **long-duration exclusive locks**
  - READ UNCOMMITTED: no read locks
  - READ COMMITTED: short duration read locks
  - REPEATABLE READ:
    - Long duration read locks on individual items
  - SERIALIZABLE:
    - All locks long duration and lock predicates
- **Trade-off: consistency vs concurrency**
- Commercial systems give choice of level

Isolation Levels in SQL

1. "Dirty reads"
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
2. "Committed reads"
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED
3. "Repeatable reads"
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
4. Serializable transactions
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

ACID

Choosing Isolation Level

- Trade-off: efficiency vs correctness
- DBMSs give user choice of level

Always read docs!

Beware!!

- Default level is often NOT serializable
- Default level differs between DBMSs
- Some engines support subset of levels!
- Serializable may not be exactly ACID

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” READ and WRITE locks
  - Strict 2PL

This is not serializable yet !!!

Why ?

4. Isolation Level Serializable

- 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
```

Can improve performance

Advanced Topics

- Aries recovery manager
- Timestamp-based concurrency control

Terminology

- **STEAL or NO-STEAL**
  - Can an update made by an uncommitted transaction overwrite the most recent committed value of a data item on disk?
- **FORCE or NO-FORCE**
  - Should all updates of a transaction be forced to disk before the transaction commits?
- Easiest for recovery: NO-STEAL/FORCE
- Highest performance: STEAL/NO-FORCE

Write-Ahead Log Revised

- Enables the use of STEAL and NO-FORCE
- **Log: append-only file containing log records**
- After a system crash, use log to:
  - Redo some transaction that did commit
  - Undo other transactions that didn't commit

Types of Logs

- Physical log: element = disk page
- Logical log: element = record
- Physiological log: combines both

Rules for Write-Ahead Log

- All log records pertaining to a page are written to disk before the page is overwritten on disk
- All log records for transaction are written to disk before the transaction is considered committed
  - Why is this faster than FORCE policy?
- **Committed transaction**: transactions whose commit log record has been written to disk

ARIES Recovery Manager

- A redo/undo log
- Physiological logging
  - Physical logging for REDO
  - Logical logging for UNDO
- Efficient checkpointing
- Read chapter 18 in the book !

Why ?

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
  - Will serve to determine if an update needs to be redone

ARIES Data Structures

- Active Transactions Table
  - Lists all running transactions (active transactions)
  - For each txn: lastLSN = most recent update by transaction
- Dirty Page Table
  - Lists all dirty pages
  - For each dirty page: recoveryLSN (recLSN)= first LSN that caused page to become dirty
- Write Ahead Log contains log records
  - LSN, prevLSN = previous LSN for same transaction
  - other attributes

ARIES Data Structures

Dirty pages

pageID	recLSN
P5	102
P6	103
P7	101

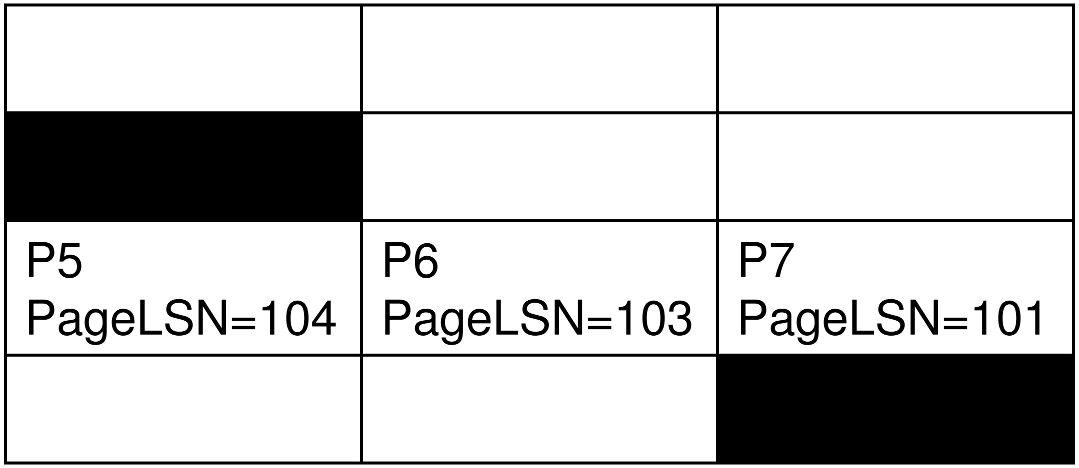
Log

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



P5 PageLSN=104

P6 PageLSN=103

P7 PageLSN=101

ARIES Method Details

Steps under normal operations:

- Transaction T writes page P
  - What do we do ?
- Buffer manager wants to evict page P
  - What do we do ?
- Transaction T wants to commit
  - What do we do ?

ARIES Method Details

Steps under normal operations:

- Transaction T writes page P
  - Update **pageLSN**, **lastLSN**, **recLSFN**
- Buffer manager wants to evict page P
  - Flush log up to **pageLSN**
- Transaction T wants to commit
  - Flush log up to current COMMIT 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

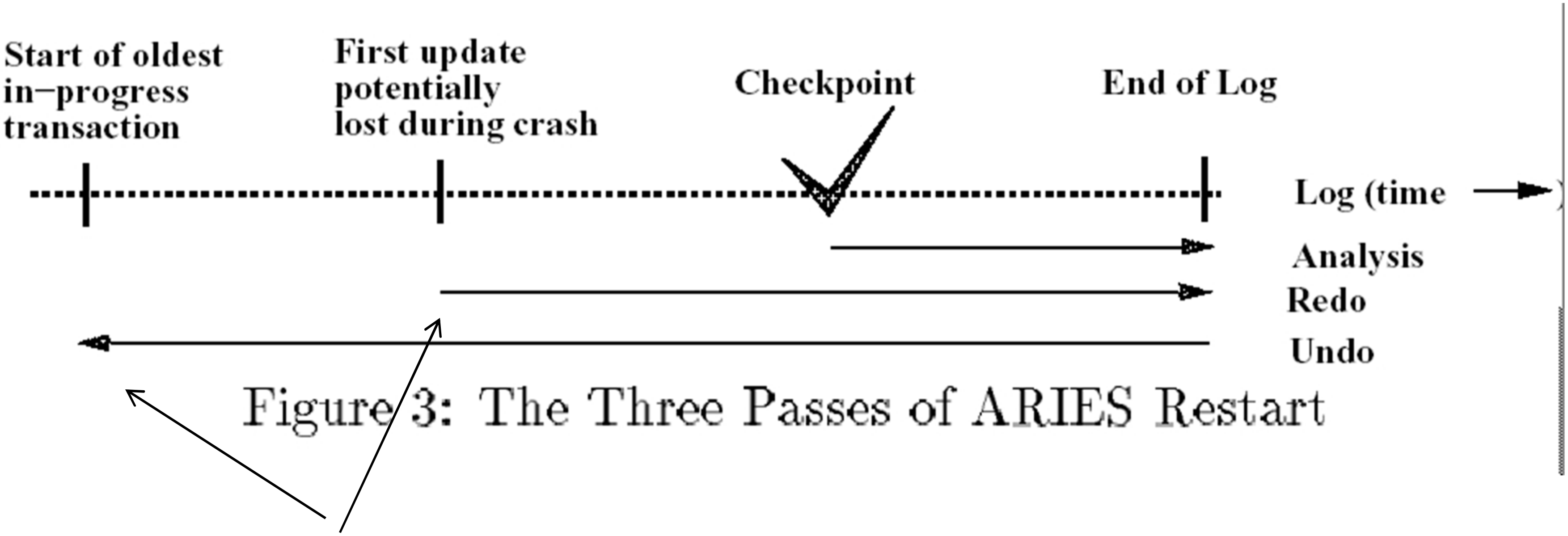


Figure 3: The Three Passes of ARIES Restart

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

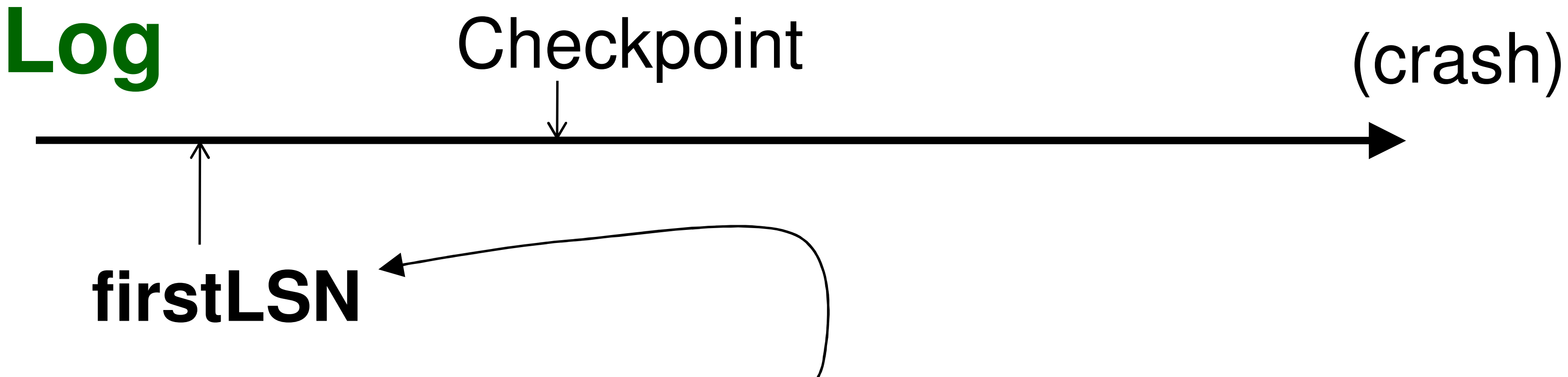
[Figure 3 from Franklin97]

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

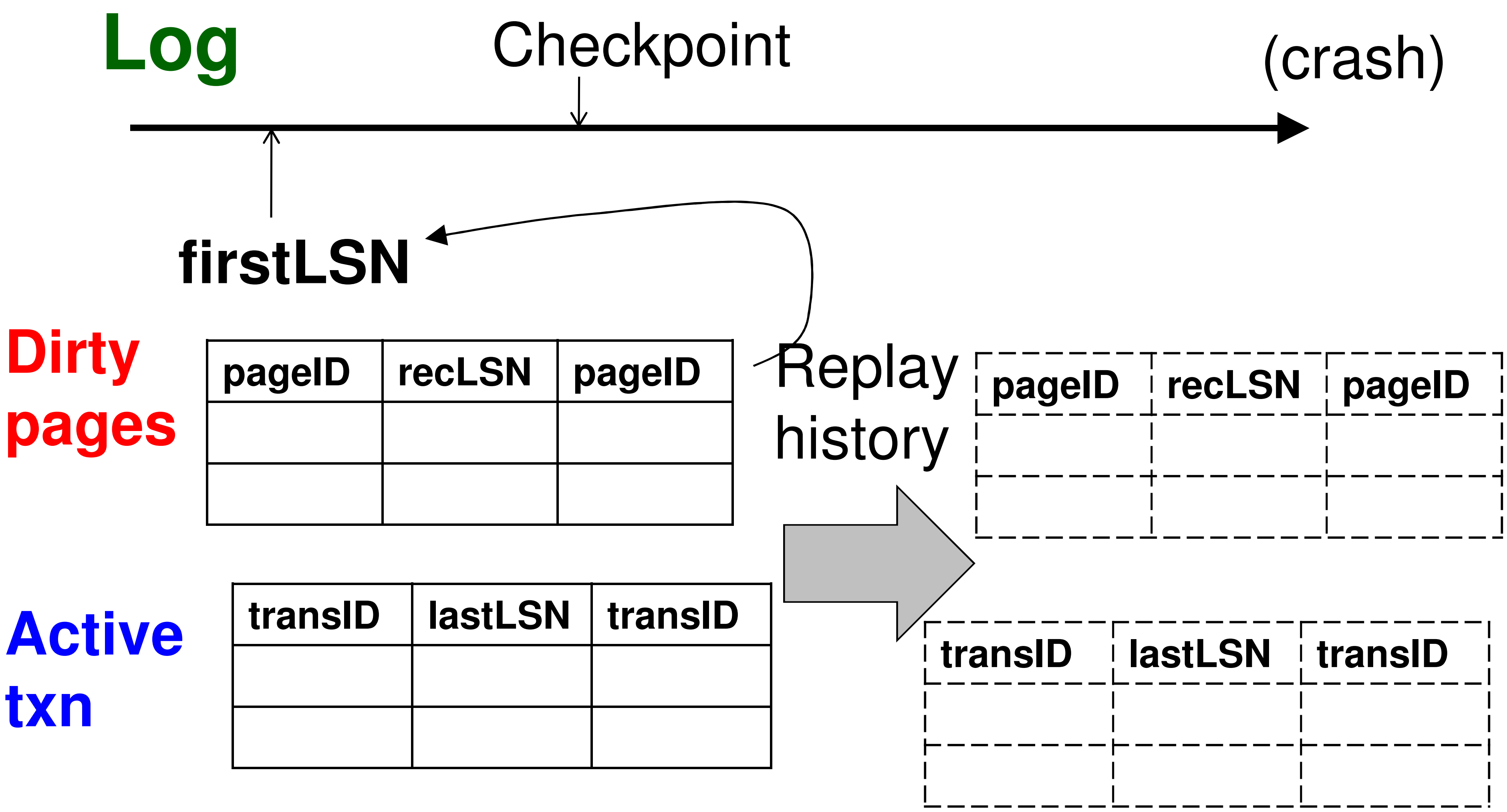
Dirty pages

pageID	recLSN	pageID

Active txn

transID	lastLSN	transID

1. Analysis Phase



Log

Checkpoint

(crash)

firstLSN

Dirty pages

pageID	**recLSN**	**pageID**

Active txn

transID	**lastLSN**	**transID**

Replay history

pageID	**recLSN**	**pageID**

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

- If affected page is not in Dirty Page Table then **do not update**
- If recoveryLSN > LSN, then **no update**
- Read page from disk;
  If **pageLSN** > LSN, then **no update**
- Otherwise perform update

3. Undo Phase

Main principle: “logical” undo

- Start from the end of the 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
- While **ToUndo** not empty:
  - Choose most recent (largest) LSN in **ToUndo**
  - If LSN = regular record: undo; 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

Handling Crashes during Undo

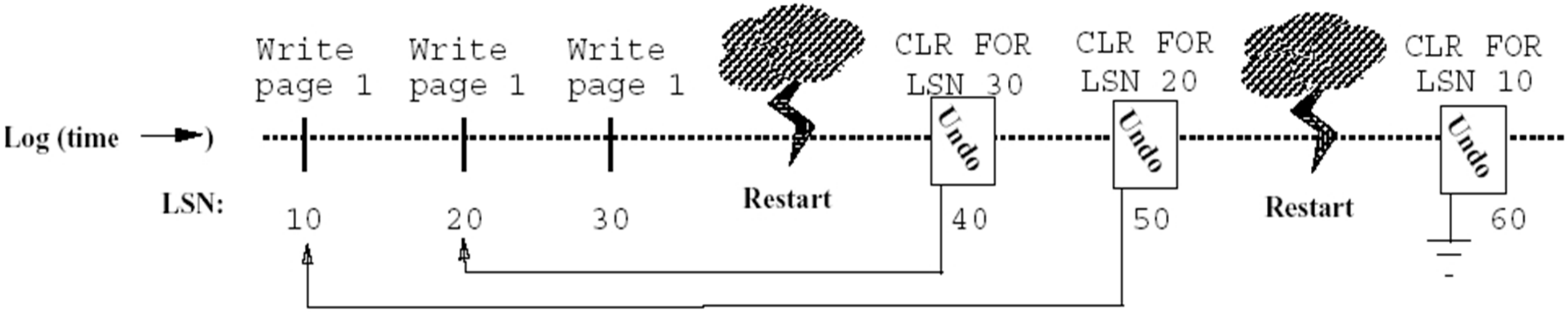


Figure 4: The Use of CLRs for UNDO

[Figure 4 from Franklin97]

Summary of Aries

- ARIES pieces together several techniques into a comprehensive algorithm
- Used in most modern database systems

Advanced 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:

In each of these cases

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

Read too late ?

Write too late ?

When T requests $r_T(X)$, need to check TS(U) <= 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 TS(T) < WT(X) then ROLLBACK

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

Transaction wants to write element X

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

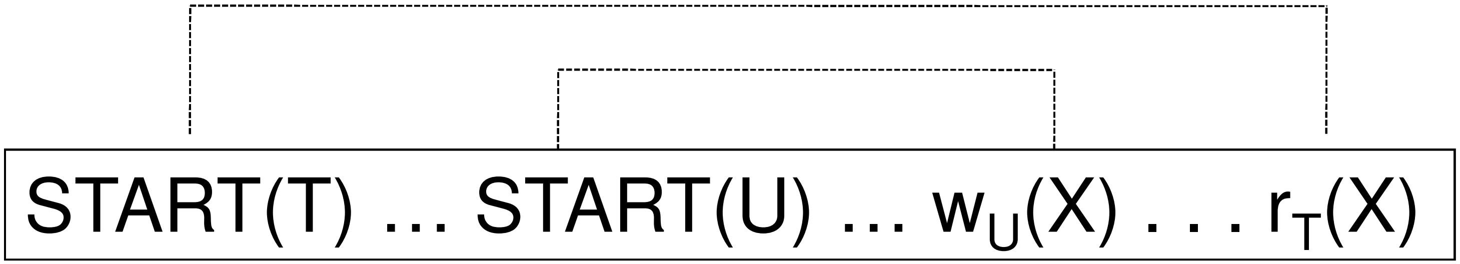
Else if TS(T) < WT(X) ignore write & continue (Thomas Write Rule)

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

Details

Read too late:

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



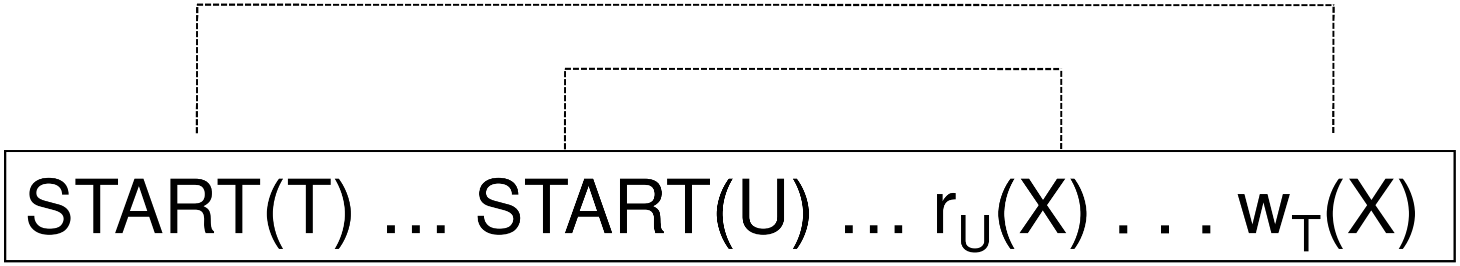
START(T) … START(U) … $w_U(X)$ . . . $r_T(X)$

Need to rollback T !

Details

Write too late:

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



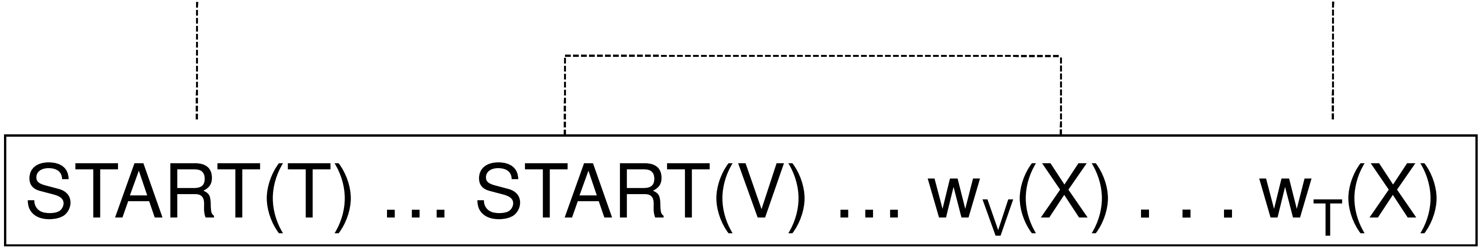
START(T) … START(U) … r_U(X) . . . w_T(X)

Need to rollback T !

Details

Write too late, but we can still handle it:

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



START(T) … START(V) … $w_V(X)$ . . . $w_T(X)$

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

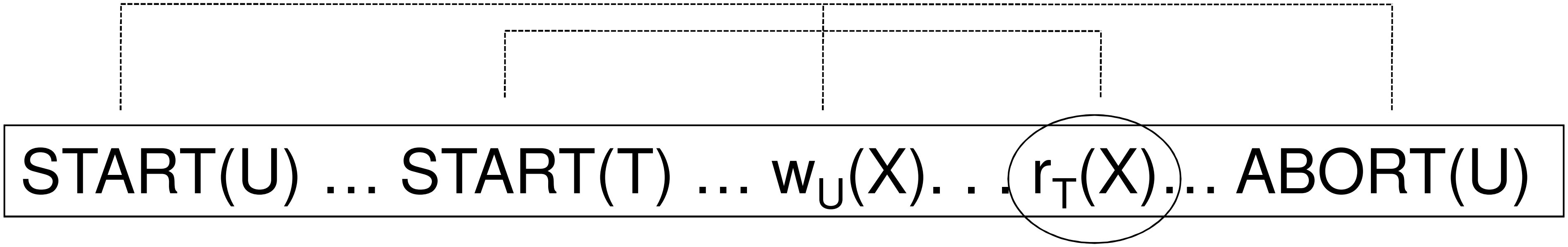
Ensuring Recoverable Schedules

- 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…



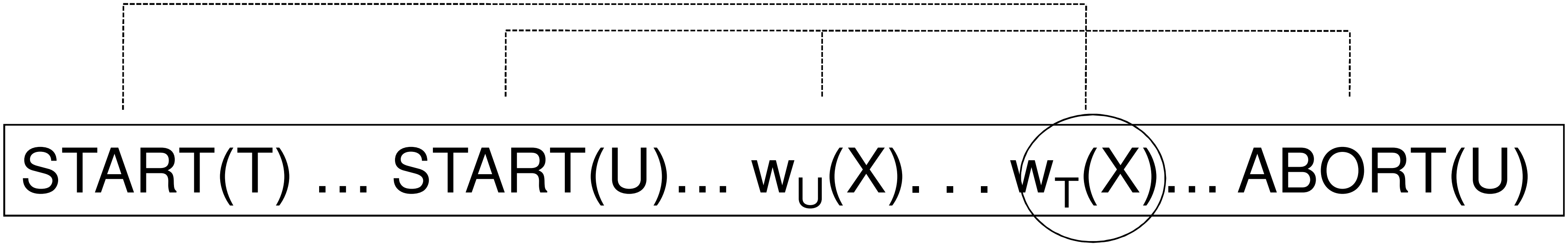
START(U) … START(T) … $w_U(X)$. . . $r_T(X)$… ABORT(U)

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

Ensuring Recoverable Schedules

Thomas' rule needs to be revised:

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



START(T) … START(U)… $w_U(X)$. . . $w_T(X)$… ABORT(U)

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

Timestamp-based Scheduling

```
Transaction wants to READ element X
    If TS(T) < WT(X)  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 TS(T) < RT(X) then ROLLBACK
    Else if TS(T) < WT(X)
        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

- Conflict-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: $X_t, X_{t-1}, X_{t-2}, \ldots$

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

- 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

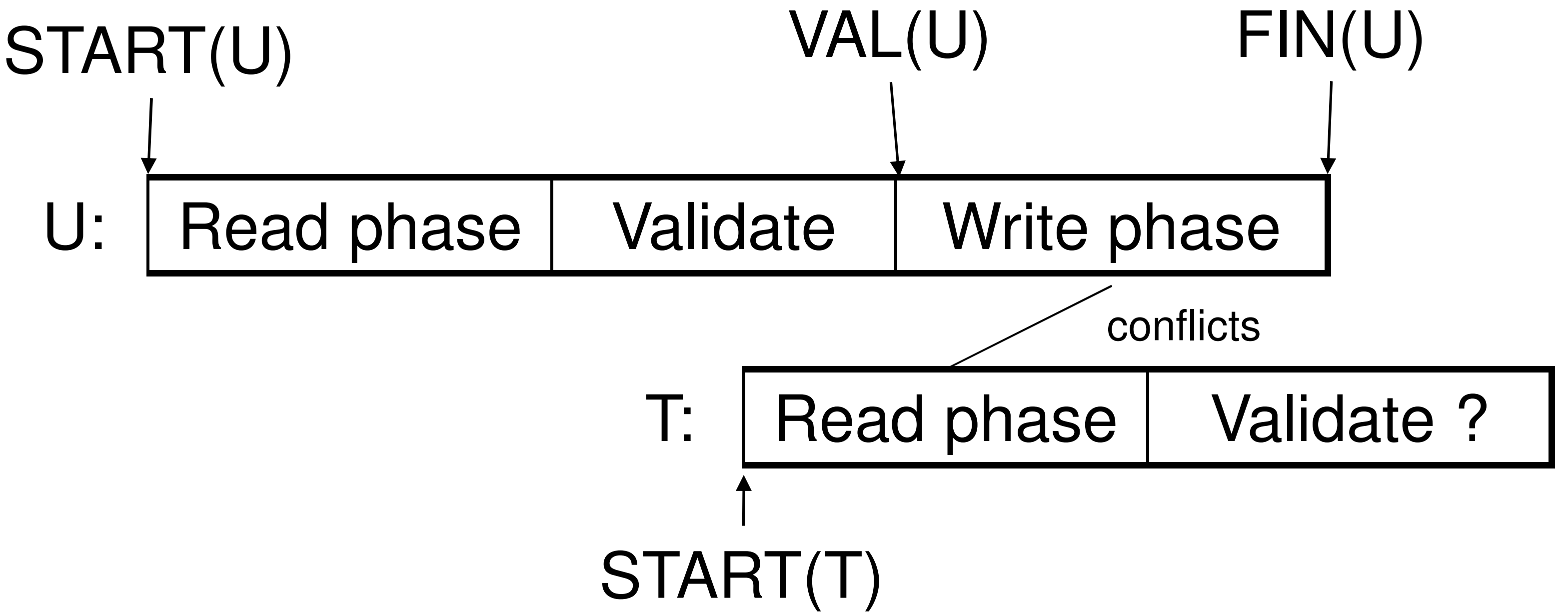
- Can delete X_t if we have a later version X_{t1} and all active transactions T have $TS(T) > t1$

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



START(U)

VAL(U)

FIN(U)

U: Read phase | Validate | Write phase

conflicts

T: Read phase | Validate ?

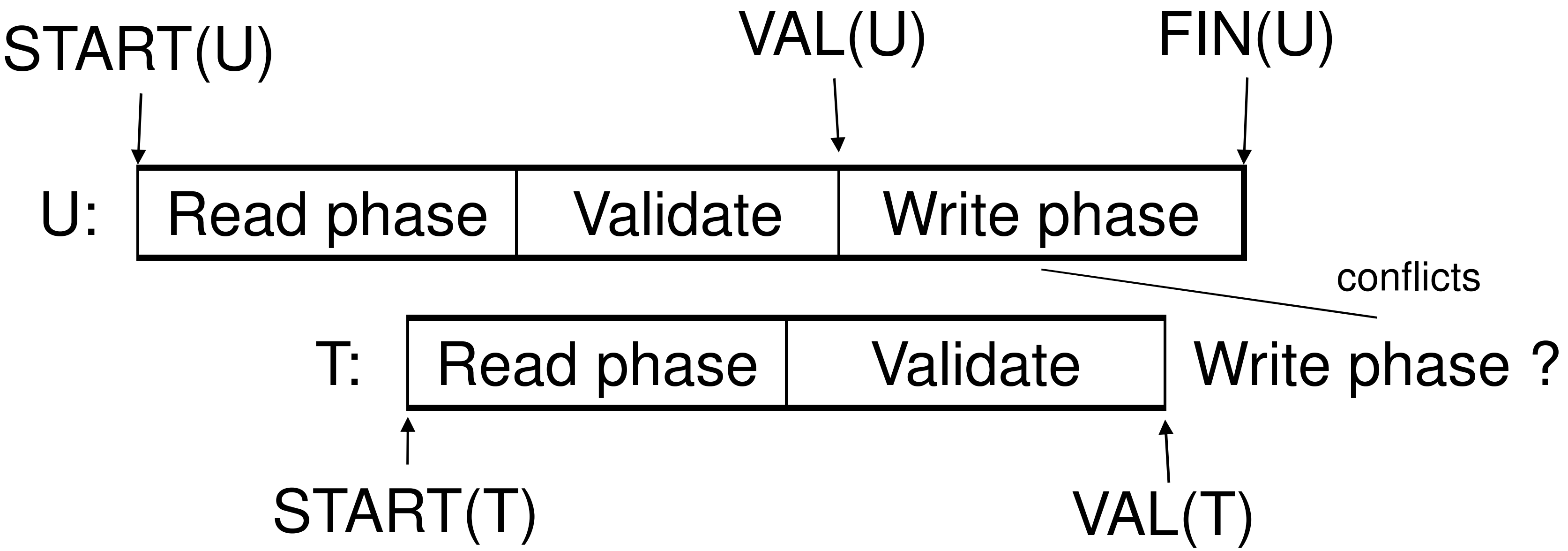
START(T)

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

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

Then ROLLBACK(T)

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



IF WS(T) $\cap$ 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: $X_{t1}, X_{t2}, X_{t3}, \ldots$
- When T reads X, return $X_{TS(T)}$.
- When T writes X: if other transaction updated X, abort
  - Not faithful to “first committer” rule, because the other transaction U might have committed after T. But once we abort T, U becomes the first committer ☺

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:
  - Multiversion concurrency control
- Oracle
  - Snapshot isolation even for SERIALIZABLE