CSE462/562: Database Systems (Spring 24)

Lecture 14: Pessimistic Concurrency Control & Crash Recovery

5/6/2024

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Last updated: 3/19/2024

Reminders

- HW6 due next Monday, 5/13, 23:59 PM EDT
    - Solution will be released immediately after
    - Grading will be finished by 5/13 EOD except for late submissions using allowed late days
- Project 5 due in two weeks, 5/20, 23:59 PM EDT
- Final reviews next week before final exam (TBD)

- Final exam: Knox 109, 5/16, 3:40 pm - 5:20 pm, please arrive 10 min earlier
    - Open-book, paper materials only, no electronics except a calculator
    - Covers Lectures 7 - 14 / HW 4 - 6
        - Relational Model, Relational Algebra & SQL
        - Query Processing
        - Query Optimization
        - Transaction, Concurrency Control & Crash Recovery

Recap on Transactions & Concurrency

- Atomicity
  - A Xact's effect is always applied as a whole, or not at all
- Consistency
  - Run by itself must leave the DB in a consistent state (no IC violations)
- Isolation
  - "protected" from the effects of concurrently scheduled other transactions
- Durability
  - If a transaction has successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk.

- Issues: Effect of interleaving transactions, and crashes, may result violate ACID.
  - Needs *concurrency control* & crash recovery

How to enforce conflict serializability?

- Two operations of two different transactions conflict if
  - Performed on the same object
  - At least one of them is a write

T1:	R_1 (A), W_1(A),		R_1(B), W_1(B)
T2:		R_2(A), W_2(A)	

Conflicts:
$R_1(A), W_2(A)$
$W_1(A), R_2(A)$
$W_1(A), W_2(A)$

- We can swap two adjacent nonconflicting operations without changing the final state

T1:	R_1 (A), W_1(A), R_1(B), W_1(B)	
T2:		R_2(A), W_2(A)

- Two schedules are conflict equivalent if one can be transformed into the other through swaps
  - Involve the same actions of the same transactions in the same order
  - Every pair of conflicting operations are ordered the same way
- Schedule S is said to be conflict serializable if it is *conflict equivalent* to some *serial* schedule S’

Pessimistic Concurrency Control

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
    - (Non-strict) 2PL Variant: Release locks anytime, but cannot acquire locks after releasing any lock.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only conflict serializable schedules.
  - Additionally, it simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing

Lock Compatibility Matrix

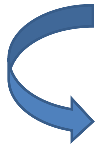
	S	**X**
S	√	–
X	–	–

Example: strict 2-PL

A

B

T1: A = A + 100, B = B - 100
T2: A = A - 100, B = B + 100



Lock upgrade

T1	T2
S(A)	
R(A)	
X(A)	
W(A)	
	request S(A) -- blocked
S(B)	
R(B)	
X(B)	
W(B)	
Commit	
Release A & B	
	S(A)
	R(A)
	X(A)
	W(A)
	……

Example: non-strict 2-PL

A

B

T1: A = A + 100, B = B - 100
T2: A = A - 100, B = B + 100

No new locks/lock upgrades at this point.

T1	T2
X(A)	
X(B)	
R(A)	
W(A)	
	request S(A) -- blocked
Release A	
	S(A)
	R(A)
	X(A)
	W(A)
R(B)	
W(B)	
Release B	
Commit	
	……

Example: non-strict 2-PL

A

B

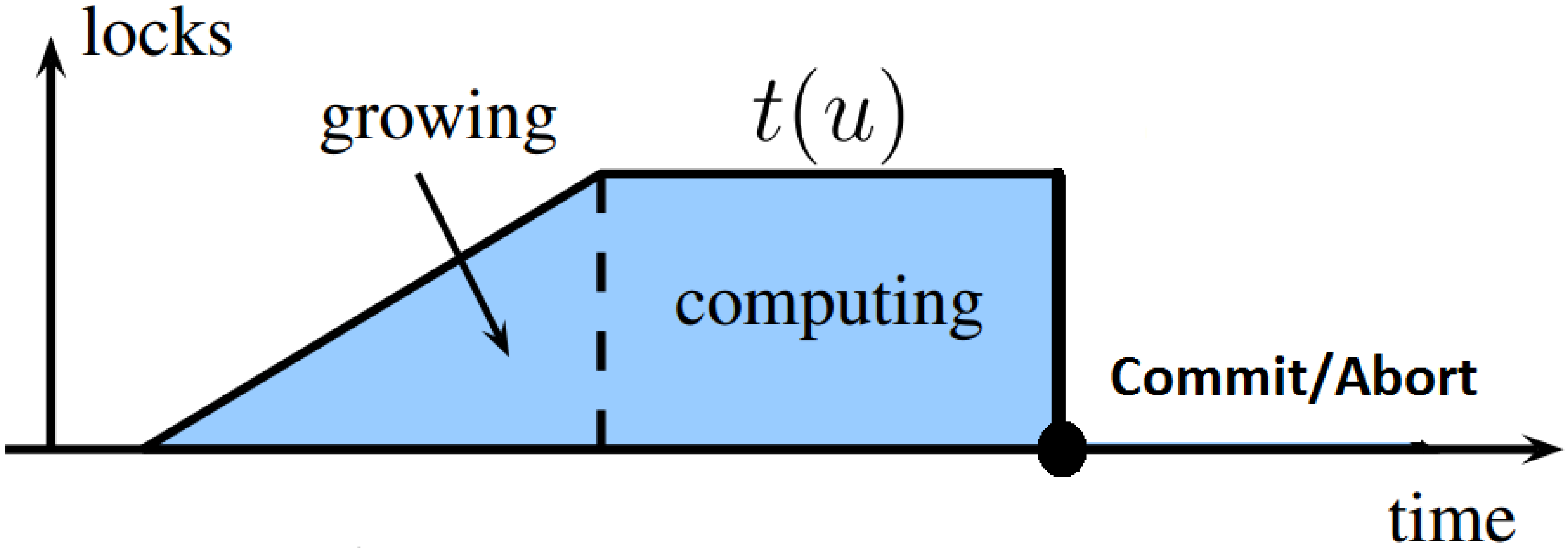
T1: A = A + 100, B = B - 100
T2: A = A - 100, B = B + 100

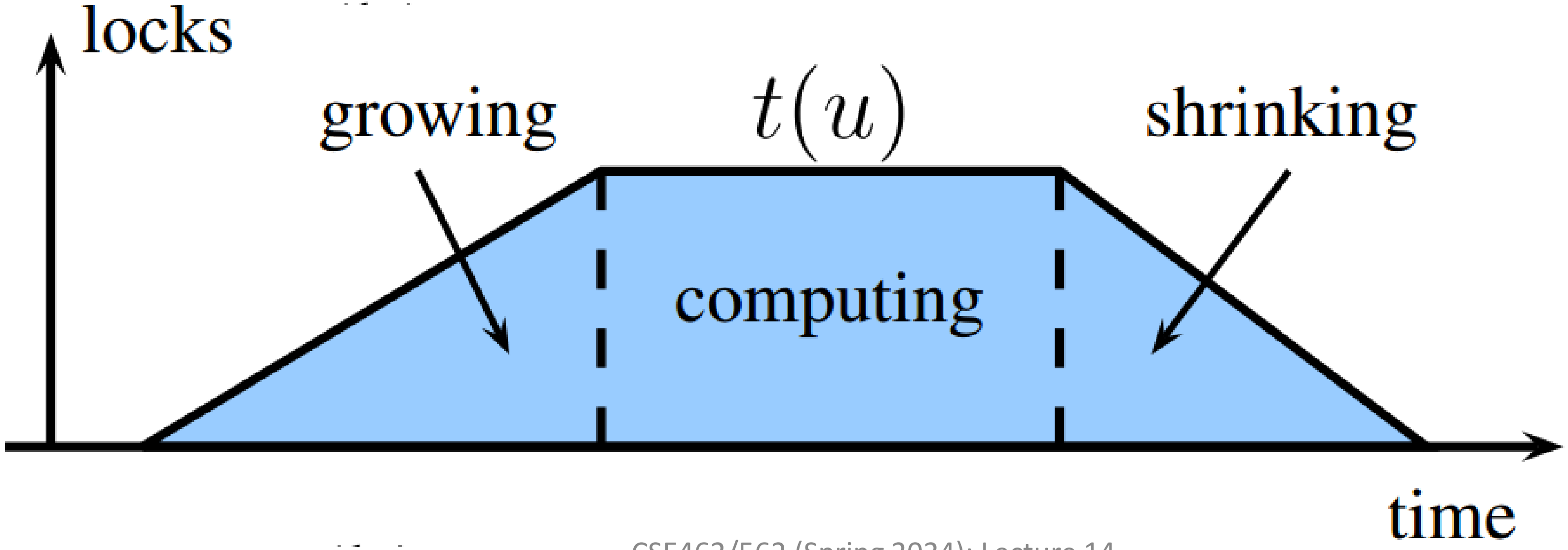
susceptible to cascading aborts!

Usually avoided in DBMS to avoid wasted work.

T1	T2
X(A)	
X(B)	
R(A)	
W(A)	
	request S(A) -- blocked
Release A	
	S(A)
	R(A)
	X(A)
	W(A)
R(B)	
W(B)	
Release B	
abort	
	abort

Strict 2-PL vs non-strict 2-PL





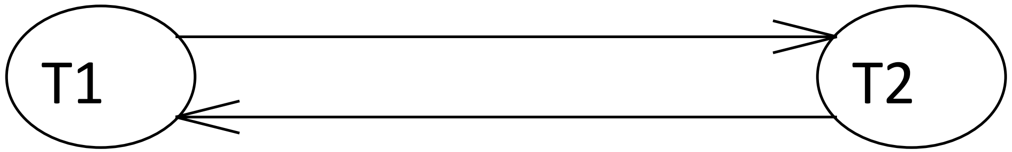
Deadlocks

A

B

T1: A = A + 100, B = B - 100
T2: B = B + 100, A = A - 100

- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Deadline ⇔ cycle in the wait-for graph
- Two ways to handle deadlocks
  - Deadlock prevention
  - Deadlock detection



T1	T2
S(A)	
R(A)	
X(A)	
W(A)	
	S(B)
	R(B)
	X(B)
	W(B)
S(B) -- blocked	
	S(A) -- blocked

Deadlock!

Deadlock prevention

- Idea: make sure wait-for graph is acyclic
  - Intuition: only allow edges to form in one of the following two directions:
    - either from older transactions to younger transactions *(wait-die)*
    - or only from younger to older *(wound-wait)*
  - Aborting a transaction prevents forming wait-for edges

- Assign priorities based on start timestamps.
  Assume Ti wants a lock that Tj holds. Two policies are possible:
  - *Wait-Die:* If Ti has lower timestamp (i.e., older) than Tj, Ti waits; otherwise Ti aborts
    - No preemption
  - *Wound-Wait:* If Ti has lower timestamp (i.e., older), Tj aborts (preempted); otherwise Ti waits
    - Preemptive scheduling

- If a transaction re-starts, make sure it gets its original timestamp
  - Why? (to avoid starvation)

Deadlock prevention: Wait-Die

A

B

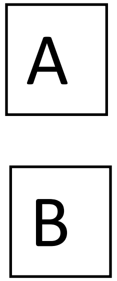
T1: A = A + 100, B = B - 100
T2: B = B + 100, A = A - 100

Wait-Die: If Ti has lower timestamp (i.e., older) than Tj, Ti waits; otherwise Ti aborts

Scenario 1: T1 requests $S(B)$ before $T2$ requests $S(A)$

T1, ts = 1	T2, ts = 2
S(A)	
R(A)	
X(A)	
W(A)	
	S(B)
	R(B)
	X(B)
	W(B)
S(B) -- blocked	
	S(A) -- abort
S(B) granted	
R(B)	
X(B)	
W(B)	
commit	
	(retry with ts = 2…)

Deadlock prevention: Wait-Die



T1: A = A + 100, B = B - 100
T2: B = B + 100, A = A - 100

Wait-Die: If Ti has lower timestamp (i.e., older) than Tj, Ti waits; otherwise Ti aborts

Scenario 2: T1 requests $S(B)$ after $T2$ requests $S(A)$

T1, ts = 1	T2, ts = 2
S(A)	
R(A)	
X(A)	
W(A)	
	S(B)
	R(B)
	X(B)
	W(B)
	S(A) -- abort
S(B) granted	
R(B)	
X(B)	
W(B)	
commit	
	(retry with ts = 2…)

Deadlock prevention: Wound-Wait

A

B

T1: A = A + 100, B = B - 100
T2: B = B + 100, A = A - 100

Wound-Wait: If Ti has lower timestamp (i.e., older), Tj aborts (preempted); otherwise Ti waits

Scenario 1: T1 requests $S(B)$ before $T2$ requests $S(A)$

T1

T2

T1, ts = 1	T2, ts = 2
S(A)	
R(A)	
X(A)	
W(A)	
	S(B)
	R(B)
	X(B)
	W(B)
S(B)	abort *(preempted)*
R(B)	
X(B)	
W(B)	
commit	
	(retry with ts = 2…)

Deadlock prevention: Wound-Wait

A

B

T1: A = A + 100, B = B - 100
T2: B = B + 100, A = A - 100

Wound-Wait: If Ti has lower timestamp (i.e., older), Tj aborts (preempted); otherwise Ti waits

Scenario 2: T1 requests $S(B)$ after $T2$ requests $S(A)$

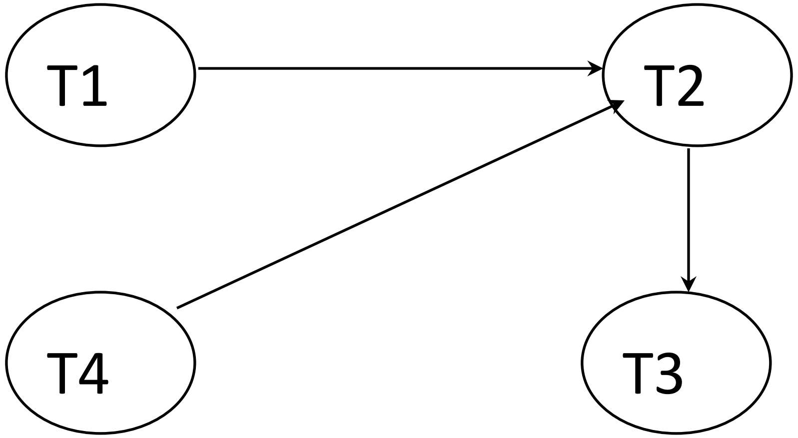
T1, ts = 1	T2, ts = 2
S(A)	
R(A)	
X(A)	
W(A)	
	S(B)
	R(B)
	X(B)
	W(B)
	S(A) -- blocked
S(B)	abort *(preempted)*
R(B)	
X(B)	
W(B)	
commit	
	(retry with ts = 2…)

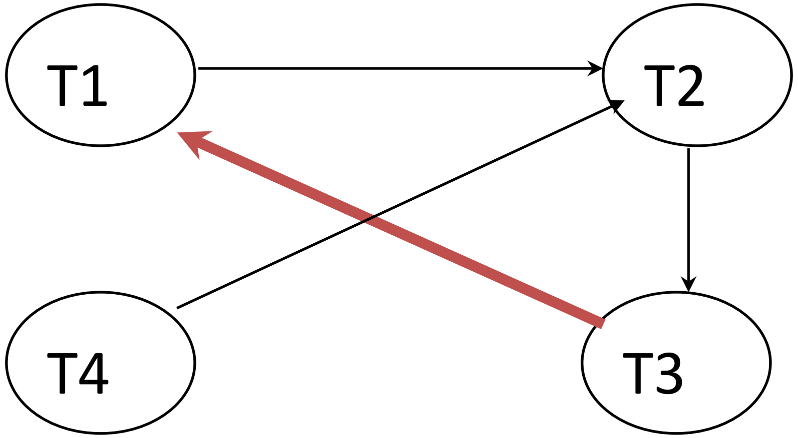
wait-for edge from T2 to T1 disappears after T2 is preempted

Deadlock detection

- Explicitly create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph
  - If there's a cycle, abort at least one transaction in the cycle

T1:	S(A), S(D),	S(B)					
T2:			X(B)		X(C)		
T3:				S(D), S(C),			X(A)
T4:						X(B)	





Deadlock detection (cont'd)

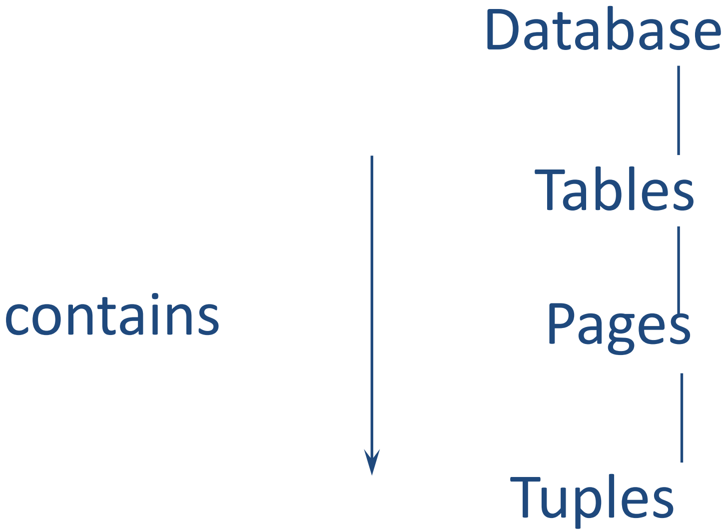
- In practice, most systems do detection
  - Experiments show that most waits-for cycles are length 2 or 3
  - Hence, only a few transactions actually need to be aborted
  - Implementations can vary
    - Can construct the graph and periodically look for cycles
      - When is the graph created ?
      - Which process checks for cycles ?
    - Can also use a "time-out" scheme
      - if T has been waiting on a lock for a long time, assume it's in a deadlock and abort

What we have glossed over

- What should we lock?
    - We assume tuples here, but that can be expensive!
    - If we do table locks, that’s too conservative
    - *Multi-granularity* locking
- How to deal with phantoms?
- Locking in indexes
    - don’t want to lock a B-tree root for a whole transaction!
    - more fine-grained concurrency control in indexes
- CC w/out locking (we’ll omit it in this course)
    - “optimistic” concurrency control
    - “timestamp” and multi-version concurrency control
    - locking usually better, though

Multi-granularity locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn't have to make same decision for all transactions!
- Data "containers" are nested:



contains

Database

Tables

Pages

Tuples

Solution: new lock modes and protocols

- Allow Xacts to lock at each level, but with a special protocol using new “intention” locks:
- Still need S and X locks, but before locking an item, Xact must have proper intension locks on all its ancestors in the granularity hierarchy.

- IS – Intent to get S lock(s) at finer granularity.
- IX – Intent to get X lock(s) at finer granularity.
- SIX mode: Like S & IX at the same time. Why useful?

	IS	IX	SIX	S	X
IS	√	√	√	√	
IX	√	√			
SIX	√				
S	√			√	
X					

Example: 2-level hierarchy

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

Tables

Tuples

	IS	IX	SIX	S	X
IS	√	√	√	√	
IX	√	√			
SIX	√				
S	√			√	
X					

Dynamic Databases – The "Phantom" Problem

- If the DB is not a fixed collection of objects, even Strict 2PL (on individual items) will not assure serializability:
- Consider T1 – "Find the highest GPA among students of each age"
    - T1 locks all pages containing sailor records with *age* = 20
        - and finds the highest GPA (say, *GPA* = 3.7).
    - Next, T2 inserts a new student; *GPA* = 4.0, *age* = 20.
    - T2 also deletes student with the highest GPA (say 3.8) among those of age = 21, and commits.
    - T1 now locks all pages containing student records with age = 21, and finds highest GPA (say, *GPA* = 3.6).
- No serial execution could lead to T1's result!

The problem

- T1 implicitly assumes that it has locked the set of all student records with *age* = 20.
  - Assumption only holds if no student records are added while T1 is executing!
  - Need some mechanism to enforce this assumption. (Index locking and predicate locking.)
- Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!
  - e.g. table locks

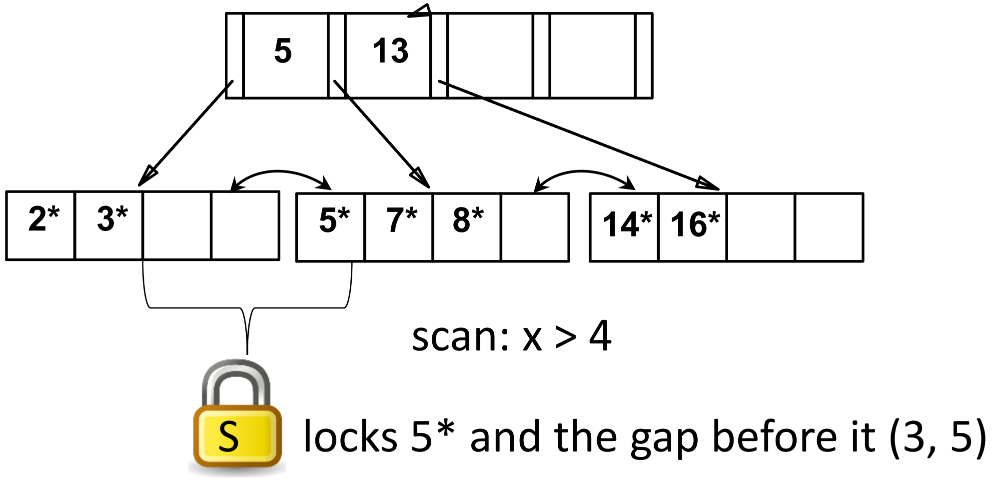
- Solution: predicate locking

Predicate locking

- Grant lock on all records that satisfy some logical predicate, e.g. *age > 2*salary*.
- Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock.
  - What is the predicate in the sailor example?
- General predicate locking has a lot of locking overhead.
  - too expensive!

Instead of predicate locking

- Full table scans lock entire tables
- Range lookups do “next-key” & gap locking
  - physical stand-in for a logical range!



scan: x > 4

locks 5* and the gap before it (3, 5)

At this point,

insert 4: blocked
insert 10?

Lock management

- Lock and unlock requests are handled by the lock manager
- Lock table: a hash table over lock table entries
  - for various resources, e.g., records, gaps, pages, tables, …
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (S, X, IS, IX, SIX)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
  - requires *latches* (e.g. reader-writer locks/semaphores), which ensure that the process is not interrupted while managing lock table entries
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock
  - Can cause deadlock problems
- Deadlock prevention/detection

Locks vs Latches

- What’s common ?
  - Both used to synchronize concurrent tasks
- What’s different ?
  - Locks are used for *logical consistency*
  - Latches are used for *physical consistency*
- Why treat ‘em differently ?
  - Latches are short-duration lower-level locks that protects critical sections in the code
    - depends on DBMS developer to prevent deadlocks
  - Locks protects data/resources, much longer duration
    - need deadlock prevention/detection, aborting transactions using priorities
    - more lock modes, hierarchical
- Where are latches used ?
  - In a lock manager !
  - In a shared memory buffer manager
  - In a B+ Tree index
  - In a log/transaction/recovery manager

Locks vs Latches

	Latches	**Locks**
Ownership	Processes	Transactions
Duration	Very short	Long (Xact duration)
Deadlocks	No detection - code carefully !	Checked for deadlocks
Overhead	Cheap - 10s of instructions (latch is directly addressable)	Costly - 100s of instructions (have to search for lock)
Modes	S, X	S, X, IS, IX, SIX
Granularity	Flat - no hierarchy	Hierarchical

Recap on Transactions & Concurrency

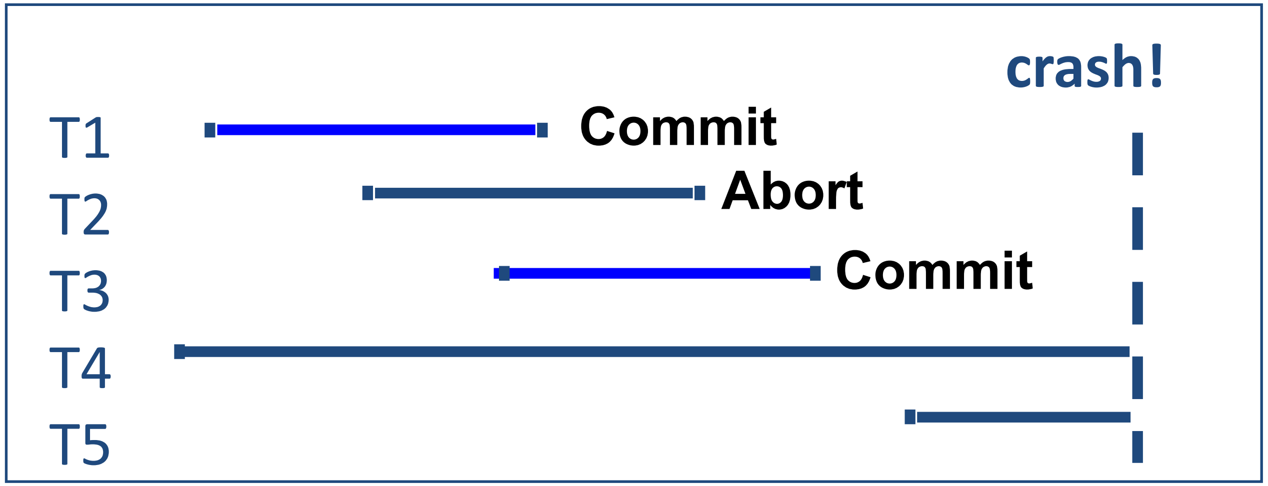
- Atomicity
  - A Xact's effect is always applied as a whole, or not at all
- Consistency
  - Run by itself must leave the DB in a consistent state (no IC violations)
- Isolation
  - "protected" from the effects of concurrently scheduled other transactions
- Durability
  - If a transaction has successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk.

- Issues: Effect of interleaving transactions, and crashes, may result violate ACID.
  - Needs concurrency control & *crash recovery*

Motivation for crash recovery

- Atomicity:
  - Transactions may abort (“Rollback”).
- Durability:
  - What if DBMS stops running? (Causes?)

- Desired state after system restarts:
  - T1 & T3 should be durable.
  - T2, T4 & T5 should be aborted (effects not seen).



Assumptions

- Concurrency control is in effect.
  - Strict 2-PL, in particular.
- Updates are happening "in place".
  - i.e. data are overwritten on (or deleted from) the actual pages.

- Can you think of a *simple* scheme (requiring no logging) to guarantee Atomicity & Durability?
  - What happens during normal execution (what is the minimum lock granularity)?
  - What happens when a transaction commits?
  - What happens when a transaction aborts?

Buffer manager plays a key role

- **Force policy –** make sure that every update is on disk before commit.
  - Provides durability without REDO logging.
  - But, can cause poor performance.

- **No Steal policy –** don't allow buffer-pool frames with *uncommited* updates to overwrite *committed* data on disk.
  - Useful for ensuring atomicity without UNDO logging.
  - But can cause poor performance.

Preferred buffer management policy: steal/no-force

- This combination is most complicated but allows for highest performance.
- *NO FORCE*: do not have to flush all dirty pages of a transaction to disk before it commits
  - complicates Durability
  - What if system crashes before a modified page written by a committed transaction makes it to disk?
  - Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.
- *STEAL:* allows buffer pool with uncommitted updates to overwrite committed data on disk
  - complicates Atomicity
  - What if the Xact that performed updates aborts?
  - What if system crashes before Xact is finished?
  - Must remember the old value of P (to support UNDOing the write to page P).

Buffer management policies

	No Steal	**Steal**
No Force		**Fastest**
Force	**Slowest**	

Performance Implications

	No Steal	**Steal**
No Force	**No UNDO** **REDO**	**UNDO** **REDO**
Force	**No UNDO** **No REDO**	**UNDO** **No REDO**

Logging/Recovery Implications

Basic Idea: Logging

- Record REDO and UNDO information, for every update, in a *log*.
  - Sequential writes to log (put it on a separate disk).
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.
- Log: An ordered list of REDO/UNDO actions
  - Log record contains:

    <XID, pageID, offset, length, old data, new data>
  - and additional control info (which we'll see soon).

Write-Ahead Logging (WAL)

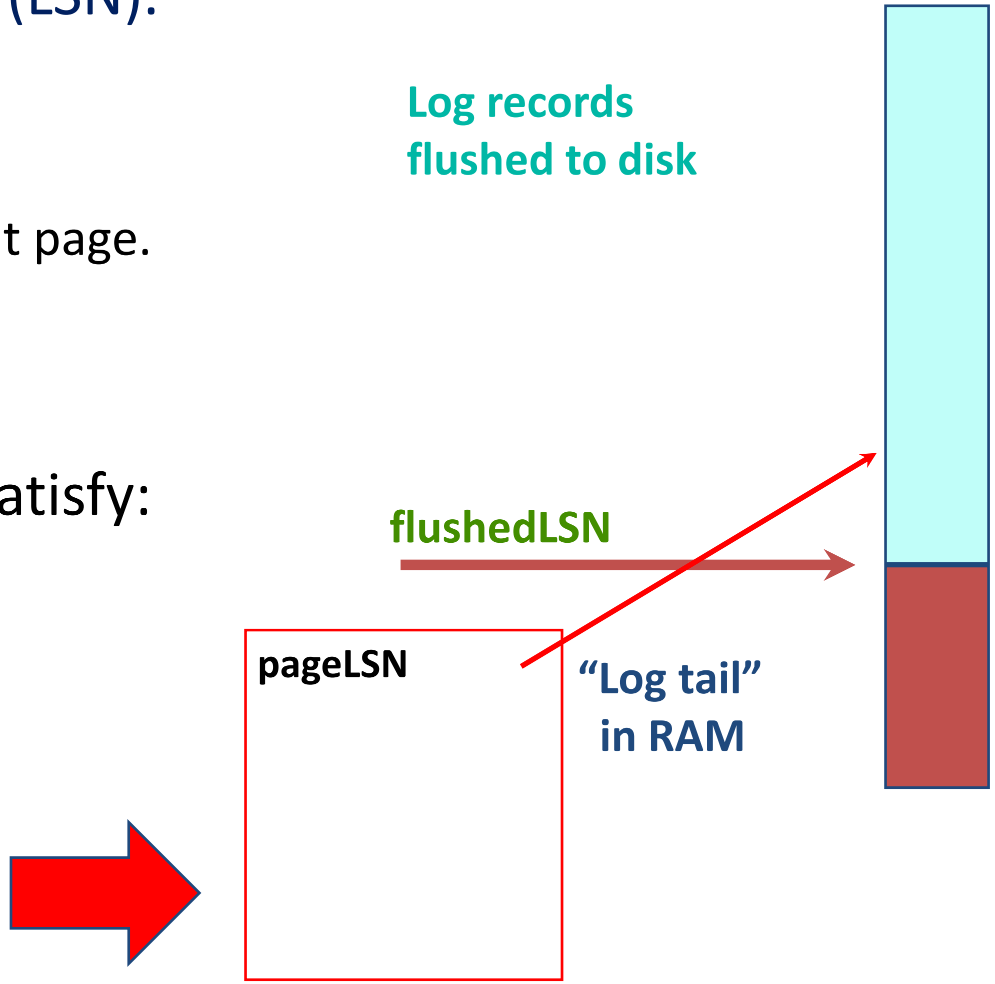
- The Write-Ahead Logging Protocol:
  - ① Must flush the log record for an update *before* the corresponding data page gets to disk.
  - ② Must flush all log records for a Xact *before commit*
    - alternatively,. transaction is not considered as committed until all of its log records including its “commit” record are on the stable log.
- #1 (with UNDO info) helps provide Atomicity.
- #2 (with REDO info) helps provide Durability.
- This allows us to employ Steal/No-Force policy

- Exactly how is logging (and recovery) done?
  - We’ll look at the ARIES algorithms.
    - Algorithms for Recovery and Isolation Exploiting Semantics

WAL & the log

- Each log record has a unique Log Sequence Number (LSN).
  - LSNs are monotonically increasing.
- Each ***data page*** contains a pageLSN.
  - The LSN of the *most recent log record* for an update to that page.
- System keeps track of flushedLSN.
  - The max LSN flushed so far.
- WAL: Before page i is flushed to disk, the log must satisfy:

$$\text{pageLSN}_i \leq \text{flushedLSN}$$



Log Records

LogRecord fields:

- LSN
- prevLSN
- XID
- type
- pageID
- length
- offset
- before-image
- after-image

update records only: pageID, length, offset, before-image, after-image

prevLSN is the LSN of the previous log record written by *this* Xact (so records of an Xact form a linked list backwards in time)

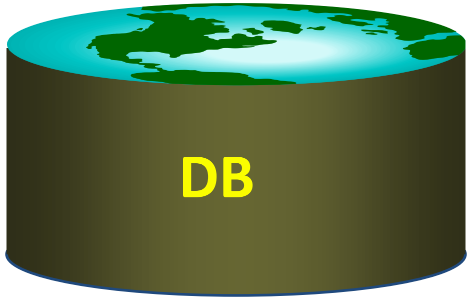
Possible log record types:

- Update
- Checkpoint (for log maintenance)
- Compensation Log Records (CLRs)
  - for UNDO actions
- Commit/Abort
- End (indicates end of commit/abort)

Other logging-related state

- Two -in-memory tables
- Transaction Table
  - One entry per currently active Xact.
    - entry removed when Xact commits or aborts
  - Contains XID, status (running/committing/aborting), and lastLSN (most recent LSN written by Xact).
- Dirty Page Table:
  - One entry per dirty page currently in buffer pool.
  - Contains recLSN -- the LSN of the log record which ***first*** caused the page to be dirty.
    - If a dirty page is flushed to disk, it is removed from dirty page table

The big picture: what’s stored and where



LogRecords

LSN
prevLSN
XID
type
pageID
length
offset
before-image
after-image

Data pages

each
with a
pageLSN

Master record

Xact Table

lastLSN
status

Dirty Page Table

recLSN

flushedLSN

Normal execution of an Xact

- Series of reads & writes, followed by commit or abort.
  - We will assume that disk write is atomic.
    - In practice, additional details to deal with non-atomic writes.
- Strict 2-PL.
- STEAL, NO-FORCE buffer management, with Write-Ahead Logging.

Transaction Commit

- Write commit record to log.
- All log records up to Xact's commit record are flushed to disk.
  - Guarantees that flushedLSN ≥ lastLSN.
  - Note that log flushes are sequential, synchronous writes to disk.
  - Many log records per log page.
- Write an end record to log (no need to flush immediately)
- Commit() returns.

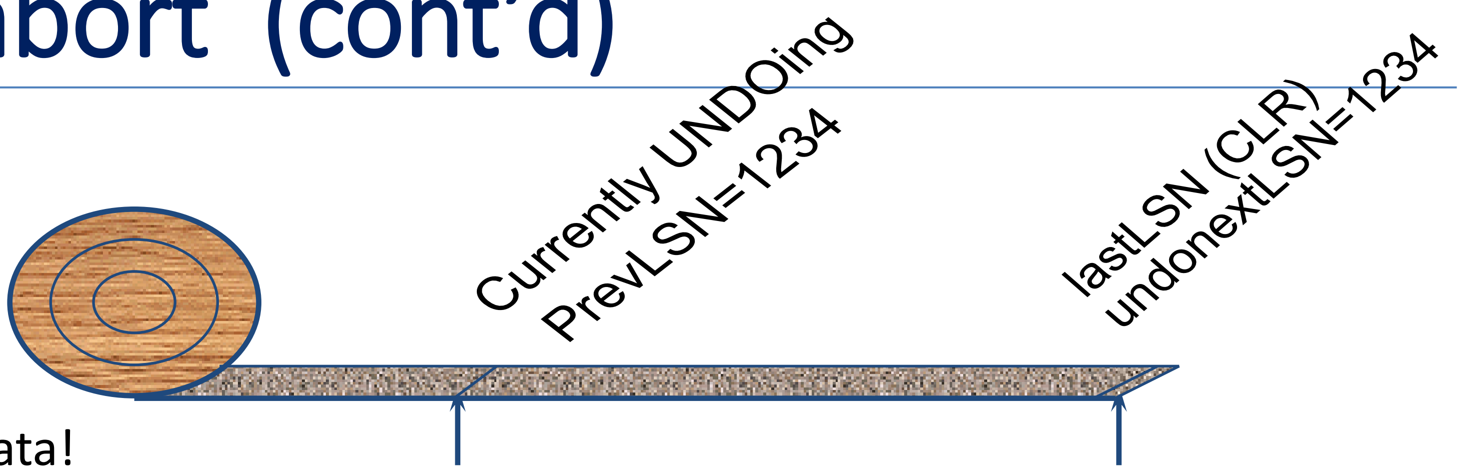
- When does a transaction becomes durable in the database?
  - When its commit log record is flushed to disk, even if there are still dirty pages in bufmgr.

Simple transaction abort

- For now, consider an explicit abort of a Xact.
  - No crash involved.
- First, set the transaction state in the transaction table to aborting.
  - Write an *Abort* log record before starting to rollback operations
- We want to "play back" the log in reverse order, UNDOing updates.
  - Get lastLSN of Xact from Xact table.
    - Can follow chain of log records backward via the prevLSN field.
  - Write a "CLR" (compensation log record) for each undone operation.
    - more details on next slide
  - Once its finished, write a transaction end log record in the disk

- Q: do we need to wait for abort, CLRs and end record to be flushed?

Simple transaction abort (cont’d)

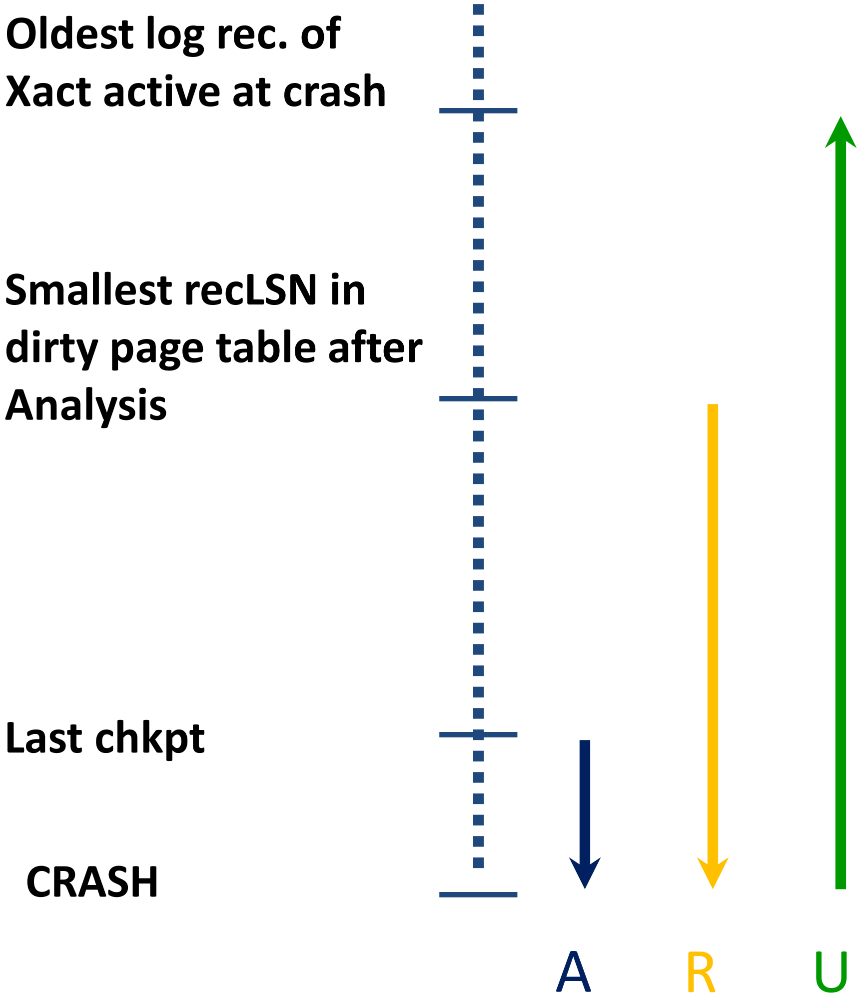


- To perform UNDO, must have a lock on data!
    - We still have the lock because of strict 2-PL.
- Before restoring old value of a page, write a CLR:
    - Must continue logging during undo in case of crash
    - CLR has one extra field: undonextLSN
        - Points to the next LSN to undo (i.e. the prevLSN of the record we’re currently undoing).
    - CLR contains REDO info
    - CLRs is *never* undone
        - Undo needn’t be idempotent (>1 UNDO won’t happen)
        - But they might be Redone when repeating history (=1 UNDO guaranteed)
- At end of all UNDOs, write an “end” log record.

Checkpointing

- Conceptually, we keep log around for all time. Obviously this has performance issues…
- Periodically, the DBMS creates a checkpoint, in order to minimize the time taken to recover in the event of a system crash. Write to log:
  - begin_checkpoint record: Indicates when chkpt began.
  - end_checkpoint record: Contains current *Xact table* and *dirty page table*. This is a `fuzzy checkpoint':
    - Other Xacts continue to run; so these tables accurate only as of the time of the begin_checkpoint record.
    - No attempt to force all dirty pages to disk; effectiveness of checkpoint limited by oldest unwritten change to a dirty page.
      - However, the more dirty page gets flushed, the shorter time will be needed in crash recovery
  - Store LSN of most recent chkpt record in a safe place (*master* record).

Crash Recovery: Big Picture



Oldest log rec. of Xact active at crash

Smallest recLSN in dirty page table after Analysis

Last chkpt

CRASH

- Start from a checkpoint (found via master record).
- Three phases. Need to do:
  - Analysis - Figure out which Xacts committed since checkpoint, which failed.
  - REDO ***all*** actions.

    (repeat history)
  - UNDO effects of failed Xacts.

Phase 1: the analysis phase

- Re-establish knowledge of state at checkpoint.
  - via transaction table and dirty page table stored in the checkpoint
- Scan log forward from checkpoint.
  - End record: Remove Xact from Xact table.
  - All Other records: Add Xact to Xact table, set lastLSN=LSN, change Xact status on commit.
  - also, for Update records: If page P not in Dirty Page Table, Add P to DPT, set its recLSN=LSN.
- At end of Analysis…
  - transaction table says which xacts were active at time of crash.
  - DPT says which dirty pages *might not* have made it to disk

Phase 2: the redo phase

- We *Repeat History* to reconstruct state at crash:
  - Reapply *all* updates (including those of aborted Xacts), redo CLRs.
- Scan forward from log rec containing smallest recLSN in DPT. Q: why start here?
- For each update log record or CLR with a given LSN, REDO the action unless:
  - Affected page is not in the Dirty Page Table, or
  - Affected page is in D.P.T., but has recLSN > LSN, or
  - pageLSN (in DB) ≥ LSN. (this last case requires I/O)
- To REDO an action:
  - Reapply logged action.
  - Set pageLSN to LSN. No additional logging, no forcing!

Phase 3: the undo phase

ToUndo={lastLSNs of all Xacts in the Trans Table}

i.e., last log entry of the aborted transactions

Repeat:

- Choose (and remove) largest LSN among ToUndo.
- If this LSN is a CLR and undonextLSN==NULL
  - Write an End record for this Xact.
- If this LSN is a CLR, and undonextLSN != NULL
  - Add undonextLSN to ToUndo
- Else this LSN is an update. Undo the update, write a CLR, add prevLSN to ToUndo.

Until ToUndo is empty.

Example of recovery

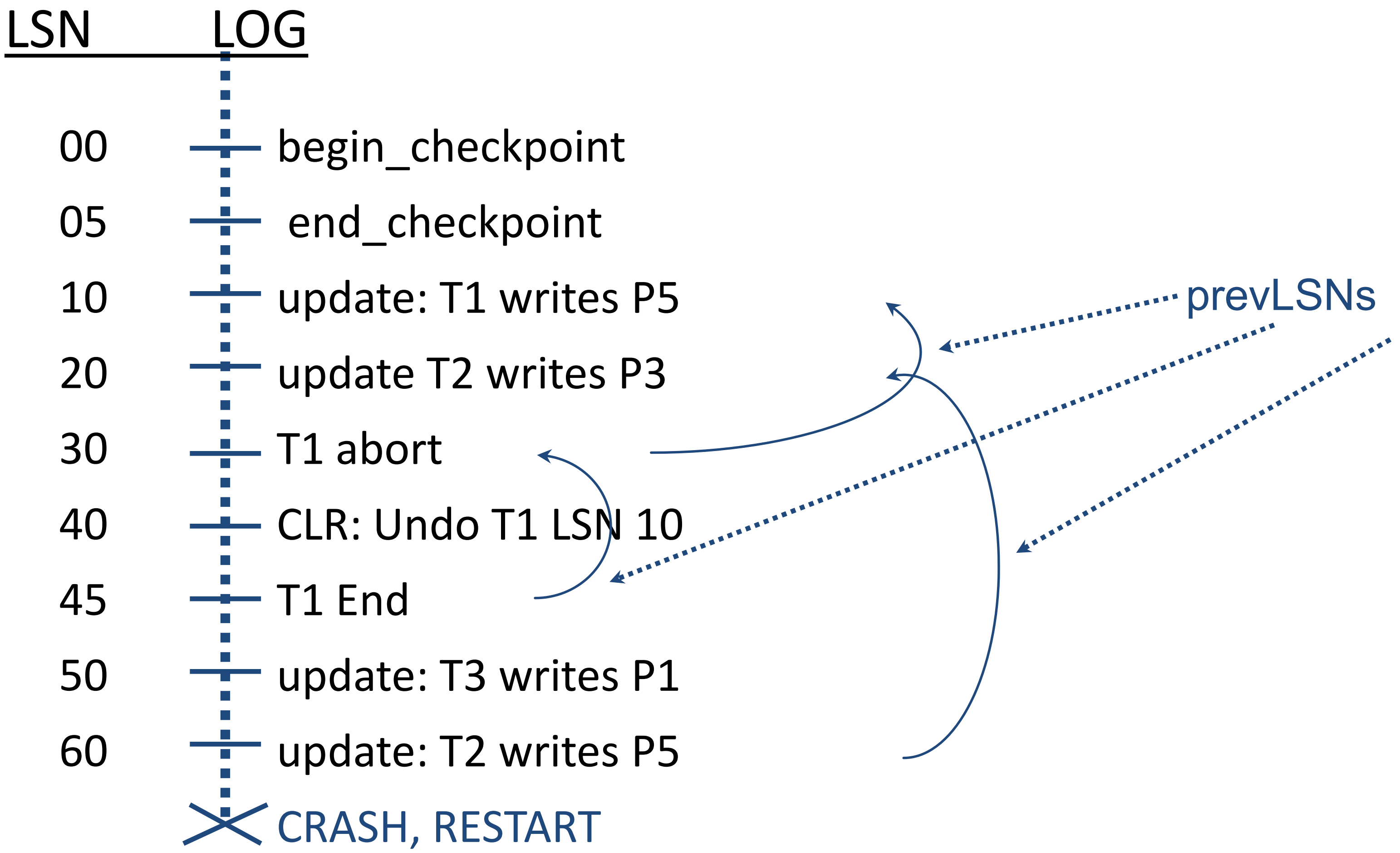
RAM

Xact Table
- lastLSN
- status

Dirty Page Table
- recLSN

flushedLSN

ToUndo



LSN	LOG
00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40	CLR: Undo T1 LSN 10
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART

Example: crash during recovery

Xact Table

lastLSN

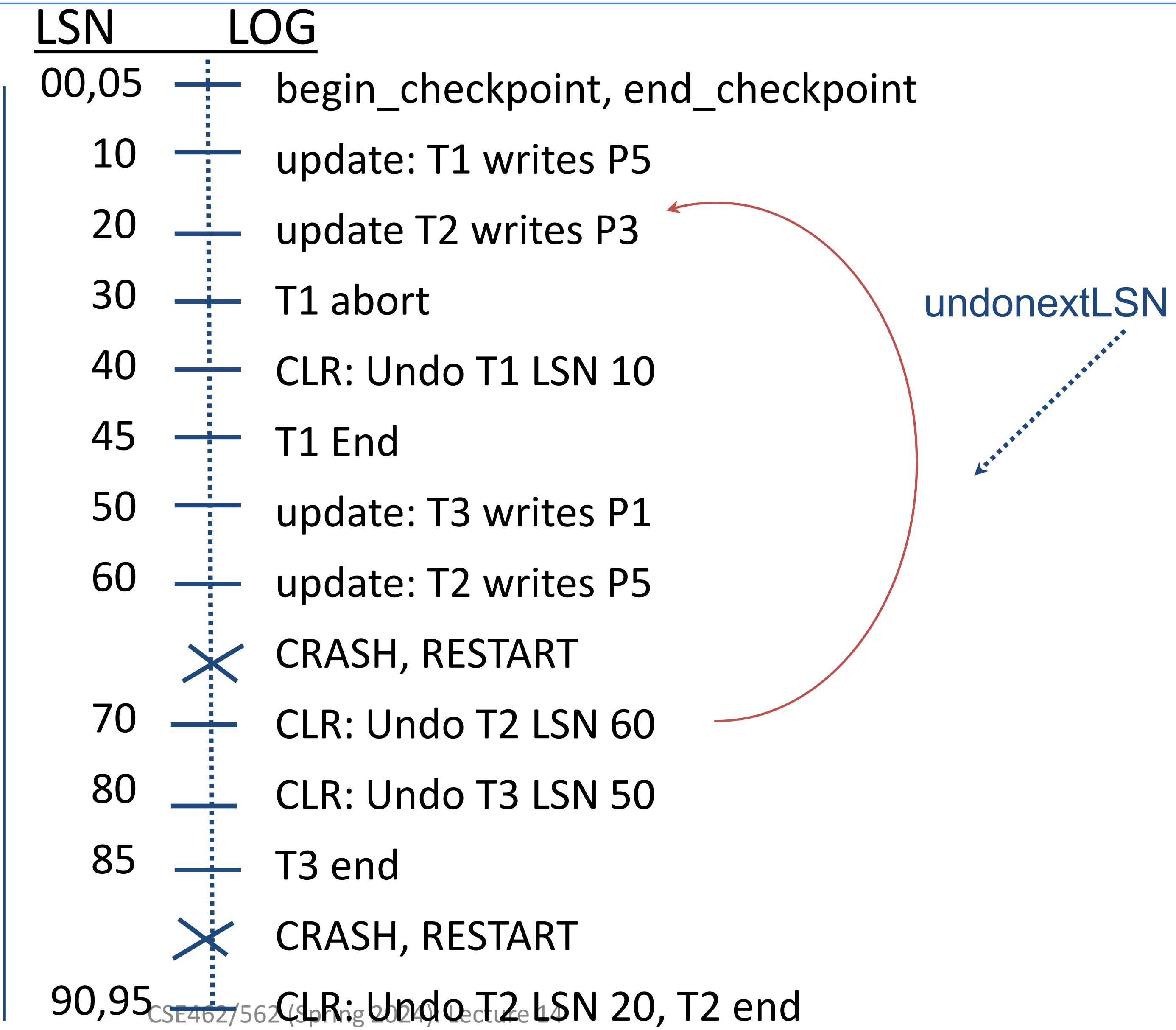
status

Dirty Page Table

recLSN

flushedLSN

ToUndo



LSN	LOG
00,05	begin_checkpoint, end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40	CLR: Undo T1 LSN 10
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART
70	CLR: Undo T2 LSN 60
80	CLR: Undo T3 LSN 50
85	T3 end
	CRASH, RESTART
90,95	CLR: Undo T2 LSN 20, T2 end

undonextLSN

Additional crash issues

- What happens if system crashes during Analysis? During REDO?
- How do you limit the amount of work in REDO?
  - Flush asynchronously in the background.
  - Watch “hot spots”!
- How do you limit the amount of work in UNDO?
  - Avoid long-running Xacts.

- What about schema changes/disk space management?

Summary of logging/recovery

- Recovery Manager guarantees Atomicity & Durability.
- Use WAL to allow STEAL/NO-FORCE w/o sacrificing correctness.
- LSNs identify log records; linked into backwards chains per transaction (via prevLSN).
- pageLSN allows comparison of data page and log records.

- Checkpointing: A quick way to limit the amount of log to scan on recovery.
- Recovery works in 3 phases:
  - Analysis: Forward from checkpoint.
  - Redo: Forward from oldest recLSN.
  - Undo: Backward from end to first LSN of oldest Xact alive at crash.
- Upon Undo, write CLRs.
- Redo “repeats history”: Simplifies the logic!

Summary

- Today
  - Pessimistic concurrency control
  - Deadlock prevention and detection
  - Crash Recovery
- Reminders
  - HW6 due next Monday, 5/13, 23:59 PM EDT
    - Solution will be released immediately after
    - Grading will be finished by 5/13 EOD except for late submissions using allowed late days
  - Project 5 due in two weeks, 5/20, 23:59 PM EDT
  - Final reviews next week before final exam (TBD)

  - Final exam: Knox 109, 5/16, 3:40 pm - 5:20 pm, please arrive 10 min earlier
    - Open-book, paper materials only, no electronics except a calculator
    - Covers Lectures 7 - 14 / HW 4 - 6
      - Relational Model, Relational Algebra & SQL
      - Query Processing
      - Query Optimization
      - Transaction, Concurrency Control & Crash Recovery