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:       | R₁(A), W₁(A), R₁(B), W₁(B) |
| T2:       | R₂(A), W₂(A)                |

Conflicts:
R₁(A), W₂(A)
W₁(A), R₂(A)
W₁(A), W₂(A)

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

| T1:       | R₁(A), W₁(A), R₁(B), W₁(B) |
| T2:       | R₂(A), W₂(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

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S</strong></td>
<td>√</td>
<td>−</td>
</tr>
<tr>
<td><strong>X</strong></td>
<td>−</td>
<td>−</td>
</tr>
</tbody>
</table>
Example: strict 2-PL

A

B

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

T1

\[ S(A) \]
\[ R(A) \]
\[ X(A) \]
\[ W(A) \]

T2

request S(A) -- blocked

\[ S(A) \]
\[ R(A) \]
\[ X(A) \]
\[ W(A) \]

Commit
Release A & B
Example: non-strict 2-PL

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

No new locks/lock upgrades at this point.
**Example: non-strict 2-PL**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A)</td>
<td></td>
</tr>
<tr>
<td>X(B)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td>request S(A) -- blocked</td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
</tbody>
</table>

Release A

R(B)

W(B)

Release B

_abort_

| S(A) |
| R(A) |
| X(A) |
| W(A) |

_abort_
Strict 2-PL vs non-strict 2-PL

- strict 2-PL
- non-strict 2-PL

Diagram showing the comparison between strict and non-strict 2-PL in terms of locks and computing over time.
**Deadlocks**

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

**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 \((wait\text{-}die)\)
    - or only from younger to older \((wound\text{-}wait)\)
  - 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

**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: $A = A + 100$, $B = B - 100$

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

**T1, ts = 1**
- $S(A)$
- $R(A)$
- $X(A)$
- $W(A)$

**T2, ts = 2**
- $S(B)$
- $R(B)$
- $X(B)$
- $W(B)$

S(B) -- blocked

S(B) granted

S(A) -- abort

(retry with ts = 2...)

CSE462/562 (Spring 2024): Lecture 14
Deadlock prevention: Wait-Die

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

T1, ts = 1
- S(A)
- R(A)
- X(A)
- W(A)

T2, ts = 2
- S(B)
- R(B)
- X(B)
- W(B)

S(B) granted
R(B)
X(B)
W(B)
commit

S(A) - abort

(retry with ts = 2...)

T1

T2
Deadlock prevention: Wound-Wait

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

**Wound-Wait:** If $T_i$ has lower timestamp (i.e., older), $T_j$ aborts (preempted); otherwise $T_i$ waits

Scenario 1: $T_1$ requests $S(B)$ before $T_2$ requests $S(A)$

- **T1, $ts = 1$**
  - $S(A)$
  - $R(A)$
  - $X(A)$
  - $W(A)$

- **T2, $ts = 2$**
  - $S(B)$
  - $R(B)$
  - $X(B)$
  - $W(B)$

  abort (*preempted*)

(retry with $ts = 2$...)
### Deadlock prevention: Wound-Wait

<table>
<thead>
<tr>
<th></th>
<th>T1, ts = 1</th>
<th>T2, ts = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(A)</td>
<td></td>
<td>S(B)</td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td>X(A)</td>
<td></td>
<td>X(B)</td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td>commit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

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

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

(retry with ts = 2...)

(T2 is preempted)

S(A) -- blocked
Deadlock detection

- Explicitly create a *waits-for graph*:
  - Nodes are transactions
  - There is an edge from $T_i$ to $T_j$ if $T_i$ is waiting for $T_j$ 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: $X(B)$
T3: $S(D)$, $S(C)$, $X(C)$
T4: $X(A)$, $X(B)$
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:
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?

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>SIX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
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
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 \textit{age} = 20
  • and finds the \textit{highest GPA} (say, \textit{GPA} = 3.7).
  • Next, T2 inserts a new student; \textit{GPA} = 4.0, \textit{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 \textit{highest GPA} (say, \textit{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 \( \text{age} = 20 \).
  • 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. $\text{age} > 2 \times \text{salary}$.
- 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!

At this point,
insert 4: blocked
insert 10?

scan: $x > 4$
locks 5* and the gap before it (3, 5)
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

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

```
crash!
```

```
T1 Commit
T2 Abort
T3 Commit
T4
T5
```
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 *uncommitted* 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).

CSE462/562 (Spring 2024): Lecture 14
Buffer management policies

<table>
<thead>
<tr>
<th>No Force</th>
<th>Force</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Steal</td>
<td>Fastest</td>
</tr>
<tr>
<td>Steal</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Performance Implications</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>No Force</th>
<th>Steal</th>
</tr>
</thead>
<tbody>
<tr>
<td>No UNDO REDO</td>
<td>UNDO REDO</td>
</tr>
<tr>
<td>No UNDO No REDO</td>
<td>UNDO No REDO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logging/Recovery Implications</th>
</tr>
</thead>
</table>

CSE462/562 (Spring 2024): Lecture 14
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

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**
- LOG
- DB

**Xact Table**
- lastLSN
- status

**Dirty Page Table**
- recLSN

**flushedLSN**

**RAM**
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 $\text{flushedLSN} \geq \text{lastLSN}$.
  - 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?
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

- 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

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>begin_checkpoint</td>
</tr>
<tr>
<td>05</td>
<td>end_checkpoint</td>
</tr>
<tr>
<td>10</td>
<td>update: T1 writes P5</td>
</tr>
<tr>
<td>20</td>
<td>update T2 writes P3</td>
</tr>
<tr>
<td>30</td>
<td>T1 abort</td>
</tr>
<tr>
<td>40</td>
<td>CLR: Undo T1 LSN 10</td>
</tr>
<tr>
<td>45</td>
<td>T1 End</td>
</tr>
<tr>
<td>50</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td>60</td>
<td>update: T2 writes P5</td>
</tr>
</tbody>
</table>

CRASH, RESTART
Example: crash during recovery

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00,05</td>
<td>begin_checkpoint, end_checkpoint</td>
</tr>
<tr>
<td>10</td>
<td>update: T1 writes P5</td>
</tr>
<tr>
<td>20</td>
<td>update T2 writes P3</td>
</tr>
<tr>
<td>30</td>
<td>T1 abort</td>
</tr>
<tr>
<td>40</td>
<td>CLR: Undo T1 LSN 10</td>
</tr>
<tr>
<td>45</td>
<td>T1 End</td>
</tr>
<tr>
<td>50</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td>60</td>
<td>update: T2 writes P5</td>
</tr>
<tr>
<td>70</td>
<td>CLR: Undo T2 LSN 60</td>
</tr>
<tr>
<td>80</td>
<td>CLR: Undo T3 LSN 50</td>
</tr>
<tr>
<td>85</td>
<td>T3 end</td>
</tr>
</tbody>
</table>

CRASH, RESTART

CLR: Undo T2 LSN 20, T2 end
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