Crash Recovery Method

Kathleen Durant
CS 3200
Lecture 11
Outline

• Overview of the recovery manager
  – Data structures used by the recovery manager
• Checkpointing
• Crash recovery
  – Write ahead logging
  – ARIES (Algorithm for recovery and isolation exploiting semantics)
Review: ACID Properties

• **Atomicity**: either the entire set of operations happens or none of it does

• **Consistency**: the set of operations taken together should move the system for one consistent state to another consistent state.

• **Isolation**: each system perceives the system as if no other transactions were running concurrently (even though odds are there are other active transactions)

• **Durability**: results of a completed transaction must be permanent - even IF the system crashes
Recovery Manager

• Recovery manager ensures the ACID principles of atomicity and durability
  – Atomicity: either all actions are done or none
  – Durability: if a transaction is committed, changes persist within the database

• Desired behavior
  – keep actions of committed transactions
  – discard actions of uncommitted transactions
Keep the committed transactions

- T3 and T4 actions should appear in the database
- T1 and T2 actions should not appear in the database
Challenges for the Recovery Manager

• Concurrency is in effect
  – Strict 2 phase locking

• Updates are happening in place
  – Overwrite of data
  – Deletion of records
Transaction

• Series of reads & writes, followed by commit or abort.
  – We will assume that write is atomic on disk.
  – In practice, additional details to deal with non-atomic writes.

• Strict 2PL.

• STEAL, NO-FORCE buffer management

• Write-Ahead Logging
Handling of the buffer pool

• FORCE – every write to disk?
  – Poor performance (many writes clustered on same page)
  – At least this guarantees the persistence of the data

• STEAL – allow dirty pages to be written to disk?
  – If so, reading data from uncommitted transactions violates atomicity
  – If not, poor performance

<table>
<thead>
<tr>
<th></th>
<th>Force - every write to disk</th>
<th>No Force – write when optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steal – use internal DB</td>
<td></td>
<td>Desired but complicated</td>
</tr>
<tr>
<td>buffer for read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Steal - always read</td>
<td>Easy but slow</td>
<td></td>
</tr>
<tr>
<td>only committed data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Complications from NO FORCE and STEAL

• NO FORCE
  – What if the system crashes before a modified page can be written to disk?
  – Write as little as possible to a convenient place at commit time to support REDOing the data update

• STEAL
  – Current updated data can be flushed to disk but still locked by a transaction T1
    • What if T1 aborts?
    • Need to UNDO the data update done by T1
Solution: Logging

• Record REDO and UNDO information, for every update, in a log.
  – Sequential writes to log (put it on a separate disk).
  – Minimal information (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
Write-ahead Logging

• The Write-Ahead Logging Protocol:
  1. Must force the log record for an update *before* the corresponding data page gets to disk.
  2. Must write all log records for a transaction *before commit*.
     – #1 guarantees Atomicity.
     – #2 guarantees Durability.

• Example: ARIES algorithm.
The Log

• Collection of records that represent the history of actions executed by the DBMS
  – Most recent portion of the log is called the log tail
  – Tail is in memory
  – Rest of the log stored of stable storage

• Actions recorded in the log:
  – Update a page
  – Commit
  – Abort
  – End
  – Undo an update
Sequencing events

• Each log record has a unique Log Sequence Number (LSN).
  – LSNs always 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 maximum LSN flushed to disk.

• WAL: Before a page is written to disk, LSN \leq \text{flushedLSN}
Tracking operations with records

- **Update a page**
  - UPDATE record is appended to the log tail
  - Page LSN of the page is set to LSN of the update record
- **Commit**
  - COMMIT type record is appended to the log with transaction id
  - Log tail written to stable storage
- **Abort**
  - ABORT record is appended to the log with the transaction id
  - Undo is initiated for this transaction
- **End**
  - After all actions are finished to complete a transaction, an END record is appended to the log
- **Undo an update**
  - When a transaction is rolled-back, its updates are undone
  - When the ‘undone’ actions are complete a compensation log record or CLR is written
Data structures associated with the log

Log sequence record
- prevLSN (links actions)
- TransactionID
- Type of action
- Length of data
- pageID
- Offset on page
- Initial value
- Final Value

Linking log to transactions
- **Transaction Table:**
  - One entry per active transaction
  - Contains Transaction ID, status (running/committed/aborted), and lastLSN.

- **Dirty Page Table:**
  - One entry per dirty page in buffer pool.
  - Contains recLSN -- the LSN of the log record which first caused the page to be dirty.
Log sequence numbers

- Every record in a log has a log sequence number to uniquely identify it **LSN**
- References to log sequence numbers in other records
  - Previous log sequence number **prevLSN**
    - Links together the log records for a transaction in the log record
  - Last sequence number **lastLSN**
    - Most recent log record for this transaction
  - Undo next sequence number **undonextLSN**
    - Found in a compensation log record (undo the operations associated with a transaction)
  - Page Log Sequence Number **pageLSN**
    - Stored in the database, one per page – it is the most recent log sequence number that changed the page
  - Recovery Log sequence Number **recLSN**
    - Stored in the dirty page table contains the first log record that caused this page to be dirty and be stored in the dirty page table
### Example of Log, Dirty Page and Transaction Table

#### Transaction Table

<table>
<thead>
<tr>
<th>TRANSId</th>
<th>lastLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1000</td>
<td>3</td>
</tr>
<tr>
<td>T2000</td>
<td>4</td>
</tr>
</tbody>
</table>

#### Dirty Page Table

<table>
<thead>
<tr>
<th>PagId</th>
<th>recLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>P500</td>
<td>1</td>
</tr>
<tr>
<td>P600</td>
<td>2</td>
</tr>
<tr>
<td>P505</td>
<td>4</td>
</tr>
</tbody>
</table>

#### LOG

<table>
<thead>
<tr>
<th>LSN</th>
<th>Prev LSN</th>
<th>TRANSId</th>
<th>type</th>
<th>pagId</th>
<th>length</th>
<th>offset</th>
<th>before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NULL</td>
<td>T1000</td>
<td>UPDATE</td>
<td>P500</td>
<td>3</td>
<td>21</td>
<td>ABC</td>
<td>DEF</td>
</tr>
<tr>
<td>2</td>
<td>NULL</td>
<td>T2000</td>
<td>UPDATE</td>
<td>P600</td>
<td>3</td>
<td>41</td>
<td>HIJ</td>
<td>KLM</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>T2000</td>
<td>UPDATE</td>
<td>P500</td>
<td>3</td>
<td>20</td>
<td>GDE</td>
<td>QRS</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>T1000</td>
<td>UPDATE</td>
<td>P505</td>
<td>3</td>
<td>21</td>
<td>TUV</td>
<td>WXY</td>
</tr>
</tbody>
</table>
Checkpointing

• 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 transactions continue to run; so these tables accurate only as of the time of the begin_checkpoint record.
• No attempt to force dirty pages to disk; effectiveness of checkpoint limited by oldest unwritten change to a dirty page. (So it’s a good idea to periodically flush dirty pages to disk!)
• Store LSN of checkpoint record in a safe place (master record).
Abort a transaction

• For now, consider an explicit abort of a transaction
  – No crash involved.
  – We want to “play back” the log in reverse order, UNDOing updates.

• Get lastLSN of transaction from the transaction table.
  – Follow chain of log records backward via the prevLSN field.

• Before starting UNDO, write an *Abort* log record.
  – For recovering from crash during UNDO!
UNDO

- To perform UNDO, must have a lock on data!
  - No problem!
- Before restoring old value of a page, write a CLR:
  - You continue logging while you UNDO!!
  - 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\textit{s never} Undone (but they might be Redone when repeating history: guarantees Atomicity!)
- At end of UNDO, write an “end” log record.
COMMIT

• Write commit record to log.
  – All log records up to Xact’s lastLSN are flushed.
  – Guarantees that flushedLSN ≥ lastLSN.

• Note that log flushes are sequential, synchronous writes to disk.
  – Many log records per log page.

• Write end record to log.
Crash recovery

• Start from a checkpoint (found via master record).

• Three phases. Need to:
  – **ANALYSIS** Determine which transactions committed since checkpoint and which ones failed
  – **REDO** *all* actions.
    • (repeat history)
  – **UNDO** effects of uncommitted transactions (the active transactions at the time of the crash)
Crash Recovery Phases

- Oldest log record of Transaction Active at crash
- Smallest recLSN In dirty page number after Analysis
- Last Checkpoint
- Crash
- Analysis
- Redo
- Undo
Analysis Phase

• Reconstruct state at latest checkpoint.
  – Get dirty page table and transaction table from end_checkpoint record.
• Scan log forward from begin_checkpoint.
  – End record: Remove transaction from transaction table.
  – Other records: Add new transaction to transaction table, set lastLSN=LSN, change transaction status on commit.
  – Update record: If P not in Dirty Page Table,
    • Add P to DIRTY PAGE TABLE, set its recLSN=LSN.
At the end of the Analysis Phase

• When Analysis phase reaches the end of log:
  – Know all transactions that were active at time of crash
  – Know all dirty pages (maybe some false positives, but that’s ok)
  – Know smallest recLSN of all dirty pages

• REDO phase has the information it needs to do its job
REDO Phase

• We repeat History to reconstruct state at crash:
  – Reapply all updates (even aborted transactions), redo CLR (compensation log record).
  – Scan forward from log record with smallest recLSN of all dirty pages. For each CLR or update log record with LSN L, REDO the action unless:
    • Affected page is not in the Dirty Page Table, or
    • Affected page is in Dirty Page Table, but has recLSN > L, or pageLSN (in DB) >= L. (need to read page from disk for this)

• To REDO an action:
  – Reapply logged action.
  – Set pageLSN to L. No additional logging!
Undo Algorithm

• Know “loser” Xacts from reconstructed Xact Table
  – Xact Table has lastLSN (most recent log record) for each Xact
• 1. $\text{ToUndo} = \{ L \mid L \text{ is lastLSN of a loser Xact} \}$
• 2. Repeat:
  – Choose largest LSN $L$ among ToUndo.
  – If $L$ is a CLR record and its undoNextLSN is NULL
    • Write an End record for this Xact.
  – If $L$ is a CLR record and its undoNextLSN is not NULL
  – Add undoNextLSN to ToUndo
  – Else this LSN is an update. Undo the update, write a CLR, add update log record’s prevLSN to ToUndo.
• 3. Until ToUndo is empty.
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.
Example

<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>15</td>
<td>update T2 writes P3</td>
</tr>
<tr>
<td>20</td>
<td>T1 abort</td>
</tr>
<tr>
<td>25</td>
<td>CLR: Undo T1 LSN 10</td>
</tr>
<tr>
<td>30</td>
<td>T1 End</td>
</tr>
<tr>
<td>35</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td>40</td>
<td>update: T2 writes P5</td>
</tr>
<tr>
<td>45</td>
<td>CRASH, RESTART</td>
</tr>
</tbody>
</table>

First write for page? Have all dirty pages? Identified all active X?
<table>
<thead>
<tr>
<th>LOG</th>
<th>Dirty Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSN</td>
<td>Pageld</td>
</tr>
<tr>
<td>10</td>
<td>NULL</td>
</tr>
<tr>
<td>15</td>
<td>NULL</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>25</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td>35</td>
<td>NULL</td>
</tr>
<tr>
<td>40</td>
<td>15</td>
</tr>
<tr>
<td>45</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transaction Table</th>
<th>Dirty Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSId</td>
<td>lastLSN</td>
</tr>
<tr>
<td>T1</td>
<td>30</td>
</tr>
<tr>
<td>T2</td>
<td>40</td>
</tr>
<tr>
<td>T3</td>
<td>35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOG</th>
<th>Transaction Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSN</td>
<td>TRANSId</td>
</tr>
<tr>
<td>10</td>
<td>NULL</td>
</tr>
<tr>
<td>15</td>
<td>NULL</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>25</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td>35</td>
<td>NULL</td>
</tr>
<tr>
<td>40</td>
<td>15</td>
</tr>
<tr>
<td>45</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Analysis Phase Example

**(LSN)**

00
05
10
15
20
25
30
35
40
45

**LOG**

begin_checkpoint
end_checkpoint
update: T1 writes P5
update T2 writes P3
T1 abort
CLR: Undo T1 LSN 10
T1 End
update: T3 writes P1
update: T2 writes P5
CRASH, RESTART

**Active Transactions**

T2
T3

**Dirty Pages**

P5 10  T1
P3 15  T2
P1 35  T3

First write for page?
Have all dirty pages?
Identified all active X?

RecLSN?
Redo Phase Example

First write for page? Have all dirty pages? Identified all active X?

<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>15</td>
<td>update T2 writes P3</td>
</tr>
<tr>
<td>20</td>
<td>T1 abort</td>
</tr>
<tr>
<td>25</td>
<td>CLR: Undo T1 LSN 10</td>
</tr>
<tr>
<td>30</td>
<td>T1 End</td>
</tr>
<tr>
<td>35</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td>40</td>
<td>update: T2 writes P5</td>
</tr>
<tr>
<td>45</td>
<td>CRASH, RESTART</td>
</tr>
</tbody>
</table>

Active Transactions
- T2
- T3

Dirty Pages
- P5 10 T1
- P3 15 T2
- P1 35 T3

RecLSN?
Undo Phase Example

First write for page?
Have all dirty pages?
Identified all active X?

Active Transactions
T2
T3

Dirty Pages
P5 10  T1
P3 15  T2
P1 35  T3

Log Sequence Number

(LSN)  LOG
00  begin_checkpoint
05  end_checkpoint
10  update: T1 writes P5
15  update T2 writes P3
20  T1 abort
25  CLR: Undo T1 LSN 10
30  T1 End
35  update: T3 writes P1
40  update: T2 writes P5
45  CRASH, RESTART

Start
Summary: Recovery Manager

- Recovery Manager guarantees Atomicity and Durability.
  - Use WAL to allow STEAL/NO-FORCE without sacrificing correctness.

- LSNs identify log records; linked into backwards chains per transaction (via prevLSN).

- pageLSN allows comparison of data page and log records
Summary

• Checkpointing: A quick way to limit the amount of log to scan on recovery.

• Recovery works in 3 phases:
  – Analysis: Walks forward from checkpoint.
  – Redo: Walks forward from oldest recLSN.
  – Undo: Walks backward from end to first LSN of oldest transaction still active at crash.