Database Management Systems 2010/11

– Chapter 8: Recovery System –

J. Gamper

- Atomicity, Durability, and Recovery System
- Log-Based Recovery
- Deferred DB Modifications
- Immediate DB Modifications
- Checkpoints
- Shadow Paging
- Recovery with Concurrent Transactions

These slides were developed by:
– Michael Böhlen, University of Zurich, Switzerland
– Johann Gamper, University of Bozen-Bolzano, Italy

Atomicity and Durability

- Recall the atomicity and durability properties of transactions
  - A transaction either completes fully with a permanent result (i.e., committed transaction)
  - or does not happen at all and has no effect on the DB (i.e., aborted/rolled-back transaction if some error occurs)
- Transactions are aborted or rolled-back if some error occurs.

Atomicity and Durability . . .

- **Failure** types:
  - **Transaction failure**
    - **Logical errors**: Do not allow a transaction to continue due to some internal condition, e.g., bad input, overflow, resource limits.
    - **System errors**: Require a DBMS to terminate an active transaction due to, e.g., a deadlock. Later re-execution is possible.
  - **System crash**: A power failure or other HW/SW failure causes the system to crash; the content of volatile storage is lost.
  - **Disk failures**: A head crash or similar disk failure destroys all or part of disk (stable) storage.

Recovery System

- **Recovery system**: Ensures atomicity and durability of transactions in the presence of failures (and concurrent transactions).
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures.
  2. Actions taken after a failure to recover the DB contents to a state that ensures atomicity, consistency and durability.

Recovery System . . .

- Problems of recovery procedures
  - The DBMS does not know which instruction was last executed.
  - Buffers may not have been written to the disk yet.
  - Observable (external) writes cannot be undone, e.g., writes to the screen or the printer. Possible solutions include:
    - Delay external writes until the end of the transaction (if possible).
    - bid external writes.
    - Relax atomicity.

Recovery System . . .

- To ensure atomicity the DBMS must first output information describing the modifications to stable storage without modifying the DB itself.
- Two approaches are studied next
  - Log-based recovery
  - Shadow paging
- In the following we also assume that transactions run serially.

Log-Based Recovery

- A **log** is the most popular structure for recording DB modifications on stable storage
  - Consists of a sequence of **log records** that record all the update activities in the DB
  - Each log record describes a significant event during transaction processing
- Types of **log records**
  - $\langle T_i, \textbf{start} \rangle$: if transaction T_i has started
  - $\langle T_i, X_j, V_1, V_2 \rangle$: before T_i executes a write(X_j), where V_1 is the old value before the write and V_2 is the new value after the write
  - $\langle T_i, \textbf{commit} \rangle$: if T_i has committed
  - $\langle T_i, \ \textbf{abort} \rangle$: if T_i has aborted
  - $\langle \textbf{checkpoint} \rangle$

Log-Based Recovery ...

- A log allows us to
  - write DB modifications to the disk
  - undo DB modifications (using the old value)
  - redo DB modifications (using the new value)
- Properties of logs
  - Logs must be placed on stable storage (before data)
  - Logs are large because they record all DB activities
  - Checkpoints are used to reduce the size of logs
    - Transactions that committed before a checkpoint don't have to be redone

Log-Based Recovery . . .

- When a **failure occurs** the following two operations can be executed:
  - **Undo**: restore DB to state prior to execution
    - **undo**(T_i) restores the value of all data items updated by transaction T_i to the old values.
    - undo must be idempotent, i.e., executing it several times must be equivalent to executing it once
  - **Redo**: perform the changes to the DB over again
    - **redo**(T_i) (re)executes all actions of transaction T_i, i.e., sets the value of all data items updated by T_i to the new values.
    - redo must be idempotent.
- Two approaches using logs
  - Deferred database modifications
  - Immediate database modifications

Deferred DB Modifications

- **Deferred DB Modification Scheme**: All DB modifications are recorded in the log but are deferred until the transaction is ready to commit (i.e., after partial commit)
- A transaction is ready to commit if the commit log-record has been written to stable storage, i.e., when transitioning to the committed state
- This schema is also known as NOUNDO/REDO

Deferred DB Modifications . . .

- Actions after a **rolled back transaction**
    - The log is ignored; nothing has to be undone
- Actions after a **crash**
    - A transaction T_i needs to be redone if and only a $\langle T_i, \mathit{start} \rangle$ and a $\langle T_i, \mathit{commit} \rangle$ record is in the log
    - To redo transactions the log has to be scanned forward.
- The old value in the log record is not needed for deferred DB updates.

Deferred DB Modifications . . .

- **Example:** Transactions T_0 and T_1 (T_0 executes before T_1)

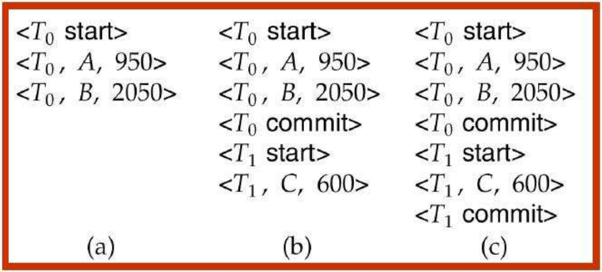
T_0 :		T_1 :	
	read(A)		**read**(C)
	$A = A - 50$		$C = C - 100$
	write(A)		**write**(C)
	read(B)		
	$B = B + 50$		
	write(B)		

- Possible order of actual outputs to the log and the DB

Log	**DB**
$\langle T_0, \textbf{start} \rangle$	
$\langle T_0, A, 950 \rangle$	
$\langle T_0, B, 2050 \rangle$	
$\langle T_0, \textbf{commit} \rangle$	
	$A = 950$
	$B = 2050$
$\langle T_1, \textbf{start} \rangle$	
$\langle T_1, C, 600 \rangle$	
$\langle T_1, \textbf{commit} \rangle$	
	$C = 600$

Deferred DB Modifications . . .

- **Example (contd.)**: Consider the log after some system crashes and the corresponding recovery actions



<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>
(a)	(b)	(c)

(a) No redo actions need to be taken

(b) **redo**(T_0) must be performed since $\langle T_0, \textbf{commit}\rangle$ is present

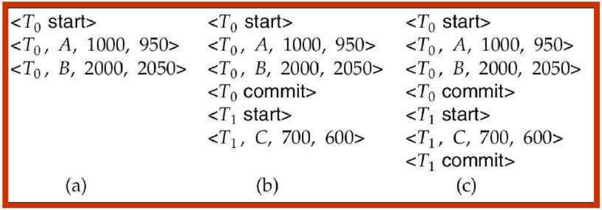
(c) **redo**(T_0) must be performed followed by **redo**(T_1) since $\langle T_0, \textbf{commit}\rangle$ and $\langle T_1, \textbf{commit}\rangle$ are present

Immediate DB Modifications

- **Immediate DB Modification Scheme**: DB modifications can be written to disk before a transaction commits. However, before doing so the modifications have to be written to the log first.
  - Known as UNDO/REDO.
- Actions after a **rolled back transaction**
  - The effects on the DB have to be undone.
- Actions after a **crash**
  - Transaction T_i needs to be **undone** if the log contains a $\langle T_i, \textbf{start} \rangle$ record, but does not contain a $\langle T_1, \textbf{commit} \rangle$ record
    - for undo the log must be scanned backwards
  - Transaction T_i needs to be **redone** if the log contains the record $\langle T_1, \textbf{start} \rangle$ and $\langle T_1, \textbf{commit} \rangle$
    - for redo the log must be scanned forwards
  - undo must be done before redo

Immediate DB Modifications . . .

- **Example (contd.):** Consider the log after some system crashes and the corresponding recovery actions



<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

(a) **undo**(T_0): B is restored to 2000 and A to 1000

(b) **undo**(T_1) and **redo**(T_0): C is restored to 700, and then A and B are set to 950 and 2050, respectively

(c) **undo**(T_0) and **redo**(T_1): A and B are set to 950 and 2050, respectively; then C is set to 600

Checkpoints

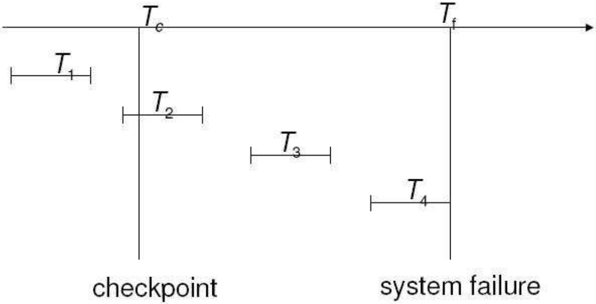
- Problems in recovery procedure
  - Searching the entire log is time-consuming
  - We might unnecessarily redo transactions which have already output their updates to the DB
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record $\langle \textbf{checkpoint} \rangle$ onto stable storage
- Any transaction T_i with a $\langle T_i, \textbf{commit} \rangle$ record in the log need not to be considered after a system crash

Checkpoints . . .

- **Recovery procedure**: Only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i need to be considered.
    1. Scan backwards from end of log to find the most recent $\langle$**checkpoint**$\rangle$ record
    2. Continue scanning backwards till a record $\langle T_i, \textbf{start}\rangle$ is found.
    3. Need only consider the part of log following $\langle T_i, \textbf{start}\rangle$ record. Earlier part of log can be ignored and can be erased.
    4. Scan forward the log (starting from T_i).
    5. For all transactions T_j with no $\langle T_j, \textbf{commit}\rangle$ record, execute **undo**(T_j).
        - Done only in case of immediate modification
    6. For all transactions T_j with $\langle T_j, \textbf{commit}\rangle$ record, execute **redo**(T_j).

Checkpoints . . .

- **Example:**



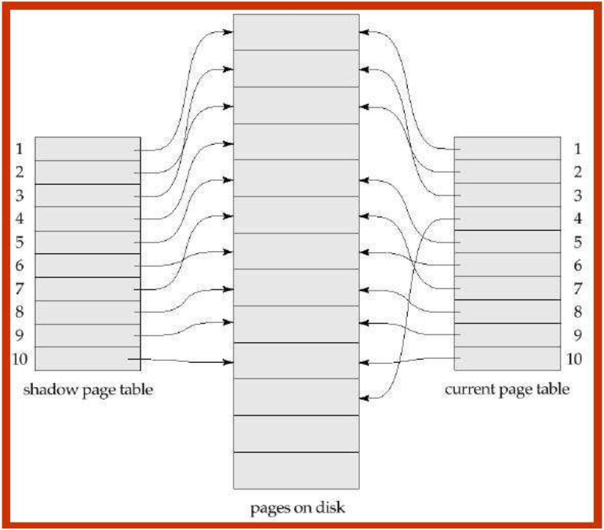
- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

Shadow Paging

- **Shadow paging** maintains two page tables during the lifetime of a transaction (page = block)
  - **Current** page table, which is typically maintained in main memory
  - **Shadow** page table, which is stored in non-volatile storage, such that state of the DB prior to transaction execution may be recovered; the shadow page table is never modified
  - Initially, both tables are identical
- Useful if transactions execute serially
- Transaction performs a **write**(X) for the first time
  - A copy of the page containing X is made onto an unused page
  - The current page table is then made to point to the copy
  - The update is performed on the copy

Shadow Paging . . .

- **Example**: Shadow and current page tables after a write to page 4



Shadow Paging . . .

- Transaction commits
  - Flush all modified pages in main memory to disk
  - Output current page table to disk
  - Make the current page table the new shadow page table
    - Keep a pointer to the shadow page table at a fixed location
    - Update the pointer to the shadow page table to point to the current page table on disk
  - Once the pointer to the shadow page table has been written, the transaction is committed

Shadow Paging . . .

- Advantages of shadow-paging over log-based recovery
  - No overhead of writing log records
  - Recovery is trivial
    - Basically, no recovery is needed after a crash
    - New transaction can start right away, using the shadow page table
- Disadvantages
  - Copying the entire page table is very expensive
    - Can be reduced by using a page table structured like a B^+-tree
  - Commit overhead is high
    - Need to flush every updated page and the page table
  - Data gets fragmented (on disk)
  - After every transaction completion garbage collection of old pages
  - Hard to extend for concurrent transactions

Recovery with Concurrent Transactions

- Extension of **log-based recovery** scheme to **concurrent transactions**
  - Assume concurrency using strict two-phase locking
    - X-locks are hold until the end of the transaction (avoids cascading)
- Logging is done as in the case for serial execution
- Checkpointing is slightly changed
  - Checkpoint log record is now of the form $\langle \textbf{checkpoint}, L \rangle$, where L is the list of transactions active at the time of the checkpoint

Recovery with Concurrent Transactions ...

- Recovery from a crash is a two-step process:
  1. Step 1: Construct an undo-list and redo-list
  2. Step 2: Perform the recovery

- **Step 1**: Construct an undo-list and redo-list
  - Initialize undo-list and redo-list to empty
  - Scan the log backwards from the end, stopping when the first $\langle \textbf{checkpoint}, L \rangle$ record is found.
    For each record found during the backward scan:
    - if the record is $\langle T_i, \textbf{commit} \rangle$: add T_i to redo-list
    - if the record is $\langle T_i, \textbf{start} \rangle$: if T_i is not in redo-list, add T_i to undo-list
  - For every T_i in L, if T_i is not in redo-list, add T_i to undo-list

Recovery with Concurrent Transactions . . .

- At this point undo-list consists of incomplete transactions which must be undone, and redo-list consists of finished transactions that must be redone.
- **Step 2**: Perform the recovery
  - Scan log backwards from most recent record, stopping when $\langle T_i, \textbf{start}\rangle$ records have been encountered for every T_i in undolist.
    - During the scan, perform **undo** for each log record that belongs to a transaction in undo-list.
  - Locate the most recent $\langle\textbf{checkpoint}, L\rangle$ record.
  - Scan log forwards from the $\langle\textbf{checkpoint}, L\rangle$ record till the end of the log.
    - During the scan, perform **redo** for each log record that belongs to a transaction on redo-list