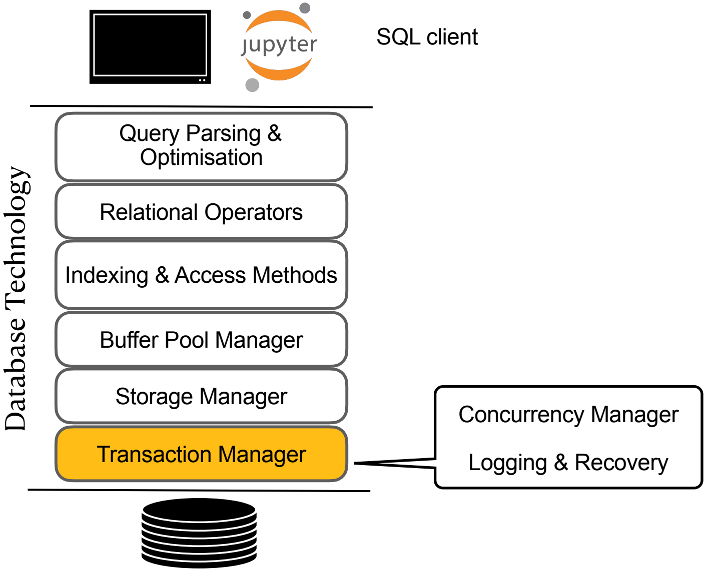
COL362/632 Introduction to Database Management Systems

Recovery

Kaustubh Beedkar

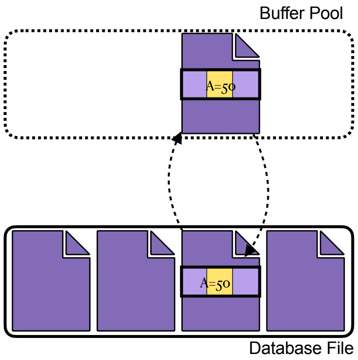
Department of Computer Science and Engineering
Indian Institute of Technology Delhi

Introduction



Recall Buffer Pool Manager

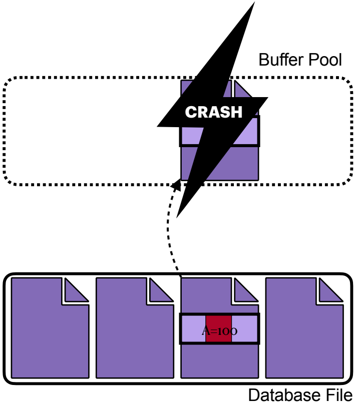
- We use Buffer Pool Manager to read and write to disk
- Actual writing to disk is dictated by the buffer replacement policy



Failure Scenario I

- Consider the following schedule

T_1
read(A)
A:=A-50
write(A)
...
commit

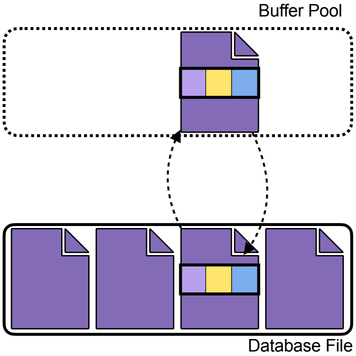


- Transaction committed before page was flushed to disk—(Durability challenges!)

Failure Scenario II

- Consider the following schedule

T_1	T_2
read(A)	
write(A)	
	read(B)
	write(B)
	commit
...	
abort	



- Changes of uncommitted Tx flushed to disk–(Atomicity challenges!)

Why do Transactions Fail?

1. Transaction Failures
   - Logical Errors (e.g., some integrity constraint violation)
   - Internal State Errors (e.g., deadlock)

2. System Failure
   - Software failures – OS or DBMS implementation (e.g., uncaught divide by zero)
   - Hardware failures (e.g., power goes off)
   - **Fail-stop assumption** – non-volatile storage contents are assumed to not be corrupted by system crash

3. Storage Media Failures
   - Disk failure (e.g., a disk head crash)
   – destruction can be detected (use checksums to detect failures)
   - Note: the database cannot recover from this! Restore from an archived version

ACID Compliance

DBMS must ensure

ACI**D**: Changes from any Tx are durable once it has been committed

ACID: No partial changes are durable if the Tx are aborted

Logging & recovery

- Actions taken during normal execution to ensure that DBMS can recover from failure
- Actions taken after a failure to recover the database to a state that ensures consistency, atomicity, and durability

Redo & Undo

The two issues

Changes from any Tx are durable once it has been committed

- **Redo** certain actions that were committed but not written to disk

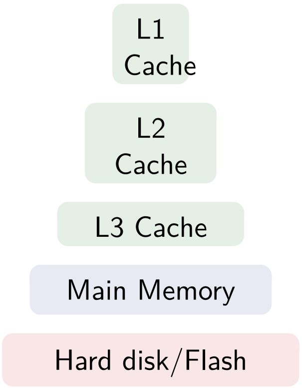
No partial changes are durable if the Tx are aborted

- **Undo** certain actions that did not commit

Storage Types

- **Volatile Storage**
  - Does not survive system crash
  - e.g., DRAM, cache memory
- **Non-volatile Storage**
  - Survives system crash
  - e.g., HDD, SSD
- **Stable Storage**
  - A mythical form of storage that survives all possible failures
  - Approximated by replication

Storage hierarchy



Some Terminology

- read(A)
- write(A)
- input(B_A)
  - Transfer the physical block B_A where the data items A resides to buffer pool
- output(B_A)
  - Transfer the buffer pool page B_A that had the data item A to disk (replace the old block B_A on disk)
  - Note: output(B_A) may not happen immediately following write(A)

In the context of transactions

- A Tx must perform read(A) before accessing A for the first time
- write(A) can be executed anytime before the transaction commits

Log-based Recovery

Key Idea

Output first the information describing the modifications to stable storage without modifying the database

Log File

- A log file contain the changes
- When a Tx T_i starts, it registers itself by writing a $\langle T_i, start\rangle$ log record
- Before T_i executes a write(A), a log record $\langle T_i, A, V_{old}, V_{new}\rangle$ is written
- When T_i finishes its last statement, the log record $\langle T_i, commit\rangle$ is written

Logging allows to perform both undo and redo operations

Database Modifications

Immediate-modifications

- Perform updates to buffer pool/disk before the Tx commits

Deferred-modifications

- Perform updates to buffer pool/disk only at the time of Tx commit

Note: All modification must be preceded by creation of a log record

Concurrency & Recovery

- Consider this schedule

T_1	T_2
...	
write(A)	
	...
	write(A)
abort	
	abort

- If T_1 aborts, we undo(T_1), and also have to undo(T_2)
- Require that if data item has been modified by a transaction, no other transaction can modify the data item until the first transaction commits or aborts
  - automatically happens in 2PL
  - time-stamp ordered protocol (recall dealing with cascaded aborts)
  - validation-based also supports this

Transaction Commits

When does a Tx go in committed state?

- when the commit log record $\langle T_i, commit \rangle$ (last record of a Tx) has been written to stable storage
- we don't care if the database has been modified or not

Also, recall partially committed state

Undo and Redo Operation

redo(T_i)

- Sets the value of all data items updated by T_i to the new values, going from first log record for T_i
- No logging is done in this case

undo(T_i)

- Restore the value of all data items updated by T_i to their old value, going backwards from the last record of T_i
  - Also write a **redo-only** record $\langle T_i, A, V_{old} \rangle$
  - When an undo(T_i) finishes, a $\langle T_i, abort \rangle$ log record is written out

Undo & Redo Operation

When recovering from failure

- T_i needs to be undone if the log
  - contains the record $\langle T_i, start \rangle$
  - but does not contain either the record $\langle T_i, commit \rangle$ or $\langle T_i, abort \rangle$
- T_i needs to be redone if the log
  - contains the records $\langle T_i, start \rangle$
  - and either the record $\langle T_i, commit \rangle$ or $\langle T_i, abort \rangle$

Example

Consider the schedule

T_1	T_2
begin	
read(A)	
A:=A-50	
write(A)	
read(B)	
B:=B+50	
write(B)	
	begin
	read(C)
	C:=C-100
	write(C)

Scenario-1: System crash after write(B)

Log records

$\langle T_1 start \rangle$
$\langle T_1, A, 100, 50 \rangle$
$\langle T_1, B, 100, 150 \rangle$

Recovery action:

- undo(T_1): B is restored to 100; A to 100
- and write log records $\langle T_1, B, 100 \rangle$, $\langle T_1, A, 100 \rangle$, $\langle T_1, abort \rangle$

Example

Consider the schedule

T_1	T_2
begin	
read(A)	
A:=A-50	
write(A)	
read(B)	
B:=B+50	
write(B)	
commit	
	begin
	read(C)
	C:=C-100
	write(C)

Scenario-2: System crash after write(C)

Log records

$\langle T_1 start\rangle$
$\langle T_1, A, 100, 50\rangle$
$\langle T_1, B, 100, 150\rangle$
$\langle T_1, commit\rangle$
$\langle T_2, start\rangle$
$\langle T_2, C, 150, 50\rangle$

Recovery action:

- redo(T_1) and undo(T_2): A is set to 50; B to 150, and C is restored to 150
- and write log records $\langle T_2, C, 150\rangle$, $\langle T_2, abort\rangle$

Example

Consider the schedule

T_1	T_2
begin	
read(A)	
A:=A-50	
write(A)	
read(B)	
B:=B+50	
write(B)	
commit	
	begin
	read(C)
	C:=C-100
	write(C)
	commit

Scenario-3: System crash after T_2 commits

Log records

$\langle T_1 start\rangle$
$\langle T_1, A, 100, 50\rangle$
$\langle T_1, B, 100, 150\rangle$
$\langle T_1, commit\rangle$
$\langle T_2, start\rangle$
$\langle T_2, C, 150, 50\rangle$
$\langle T_2, commit\rangle$

Recovery action:

- redo(T_1) and redo(T_2): A is set to 50; B to 150. Then C is set to 50

Checkpoints

- Redo and Undo operations can have significant overhead
  - Processing entire log can be time consuming
  - Redoing Tx that have already written to disk is wasteful
- **Checkpoints**
  - Special marker: recovery only needs to look at parts of the log after checkpoint and just before the checkpoint
- Checkpointing is done periodically
  - output all log records from buffer pool to stable storage
  - output all modified buffer blocks to disk
  - Write a log record $\langle checkpoint, L \rangle$, where L is a list of all transactions active at the time of checkpoint

    No Tx allowed to update while checkpointing is in progress

Write-Ahead Logging

Key Ideas

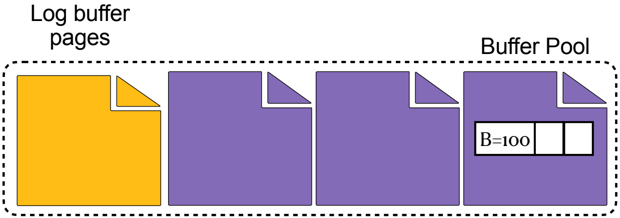
- Tx T_i enters commit state only after $\langle T_i, commit \rangle$ log record has been output to disk
- Before the $\langle T_i, commit \rangle$ log record can be output to stable storage, all log records must have been output to stable storage
- Write to disk the log file records corresponding to database modifications **before** the buffer pool manager flushes pages to disk
- Buffer Pool Policy: Steal + No-force
  - Steal policy: Pages of uncommitted Tx may be written to disk
  - No-force policy: Allow Tx to commit even it if it has modified pages that have not been written to disk

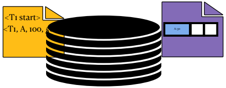
Write-Ahead Logging

Consider the schedule

T_1
begin
read(A)
write(A)
read(B)
write(B)
...
commit

- Assume that the modified page (with $A = 50$) had to be replaced
- First write the log page to disk, then flush the page

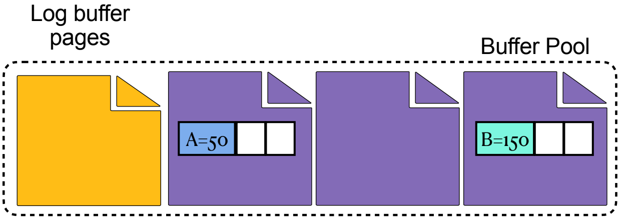




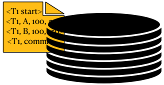
Write-Ahead Logging (scenario 2)

Consider the schedule

T_1
begin
read(A)
write(A)
read(B)
write(B)
...
commit



- Assume that T_1 is ready to commit
- First write the log page to disk, then commit the Tx



Optimization

- Group commits: batch multiple commits

Recovery Algorithm (sketch)

Write ahead logging (during normal execution)

- $\langle T_i \ start \rangle$ when the Tx begins
- $\langle T_i, \ A, \ V_{old}, V_{new} \rangle$ for each write
- $\langle T_i \ commit \rangle$ when Tx ends

Recovery Algorithm (sketch)

In case of transaction failure

assuming T_i needs to be rolled back

- Scan log backwards
- For each log record of the form $\langle T_i, X, V_{old}, V_{new} \rangle$
  - perform undo by writing V_{old} to X
  - write a **compensation log record** $\langle T_i, X, V_{old} \rangle$
- After $\langle T_i \ start \rangle$ is found, stop scanning and write log record $\langle T_i \ abort \rangle$

Recovery Algorithm (sketch)

In case of system failure

1. Redo phase: replay updates of **all** transactions
   - Find last checkpoint $\langle checkpoint\ L \rangle$ record and set undo-list to L
   - Scan log forward from $\langle checkpoint\ L \rangle$ record
     - whenever a record $\langle T_i, X, V_{old}, V_{new} \rangle$ is found, redo by writing V_{new} to X
     - whenever a record $\langle T_i\ start \rangle$ is found, add T_i to undo-list
     - whenever a record $\langle T_i\ commit \rangle$ or $\langle T_i\ abort \rangle$ is found, remove T_i from undo-list

Recovery Algorithm (sketch)

In case of system failure

2. Undo phase:
   - Scan log backwards from end
     - whenever a record $\langle T_i, X, V_{old}, V_{new} \rangle$ is found, where T_i is in undo list
       - perform undo by writing V_{old} to X
       - write a **compensation log record** $\langle T_i, X, V_{old} \rangle$
     - whenever a record $\langle T_i \ start \rangle$ is found where T_i is in undo list
       - write log record $\langle T_i \ abort \rangle$
       - remove T_i from undo-list
     - stop when undo-list is empty

Recovery Example

- Consider the following log

  $\langle T_1\ start \rangle$
  $\langle T_1, B, 100, 150 \rangle$
  $\langle T_2\ start \rangle$
  $\langle checkpoint\ \{T_1, T_2\} \rangle$
  $\langle T_2, C, 100, 200 \rangle$
  $\langle T_2\ commit \rangle$
  $\langle T_3\ start \rangle$
  $\langle T_3, A, 100, 50 \rangle$
  $\langle T_1, B, 100 \rangle$
  $\langle T_1\ abort \rangle$
  $\langle T_3, A, 100 \rangle$
  $\langle T_3\ abort \rangle$

Scan forward from checkpoint (redo phase)

- Undo-list: $T_1 \cancel{T_1}$, $T_2 \cancel{T_2}$, T_3
- After $\langle T_2\ commit \rangle$ Remove T_2 from undo-list
- After $\langle T_3\ start \rangle$ Add T_3 to undo-list
- After $\langle T_1\ abort \rangle$ Remove T_1 from undo-list

Scan backward (undo phase)

- After writing 100 to A, Write compensation log record $\langle T_3, A, 100 \rangle$
- After $\langle T_3\ start \rangle$, write $\langle T_3\ abort \rangle$