Database Recovery

CSC 443

Akshay Arun Bapat
with huge thanks to S. Sudarshan, Nick Koudas and Niv Dayan

Motivation

- Failures are common and do happen
- Recall Atomicity, Consistency, Isolation, Durability

- Transactions may want to abort
- DBMS stops running?

Want to conserve these properties even with failures

Failures

- Transaction failure
  - Logical error
  - System error
- System crash
  - Software/Hardware failure
  - Power failure
- Disk failure
  - Bad sector (part failure)
  - Head crash (complete drive failure)
- Data center failure
  - Disaster

Disk failure

- Storage disks can fail leading to data loss
- Causes
  - Head crash
  - Circuit failure
  - Motor failure
- Detectable
  - Block checksums
- Data loss
  - None
  - Partial
  - Full

RAID 4

- Block striping
- Dedicated parity disk

$$Ap = A1 \oplus A2 \oplus A3$$

$$A1 = Ap \oplus A2 \oplus A3$$

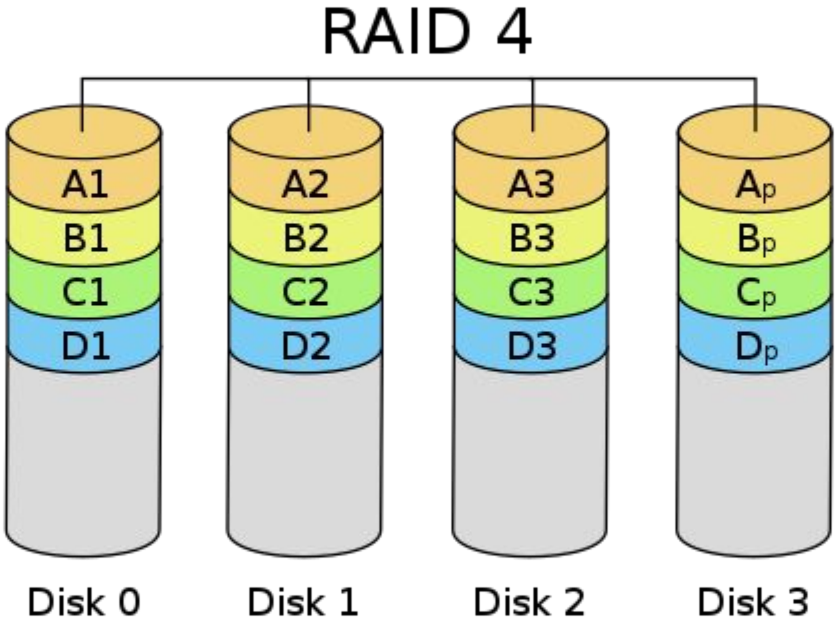


Figure from Wikipedia

RAID 5

- Block striping
- Distributed parity

$Ap = A1 \oplus A2 \oplus A3$

$A1 = Ap \oplus A2 \oplus A3$

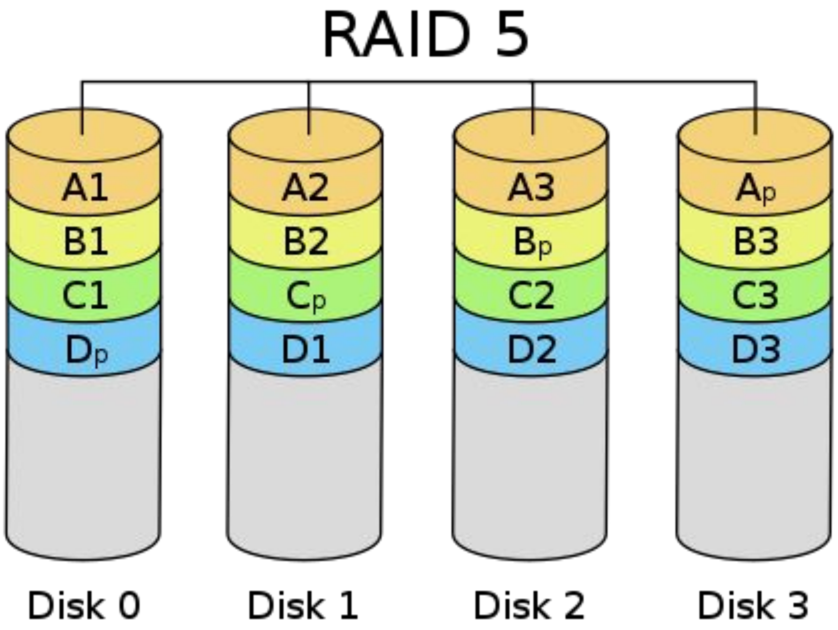


Figure from Wikipedia

RAID 4 vs RAID 5

- Sequential writes?
- Sequential reads?
- Random writes?

- Random reads?

RAID 4 vs RAID 5

- Sequential writes
  - Same
- Sequential reads
  - Same
- Random writes
  - RAID 4 has bottleneck
  - RAID 5 much better
- Random reads
  - RAID 5 better

RAID 4

- Block striping
- Dedicated parity disk

$Ap = A1 \oplus A2 \oplus A3$

$A1 = Ap \oplus A2 \oplus A3$

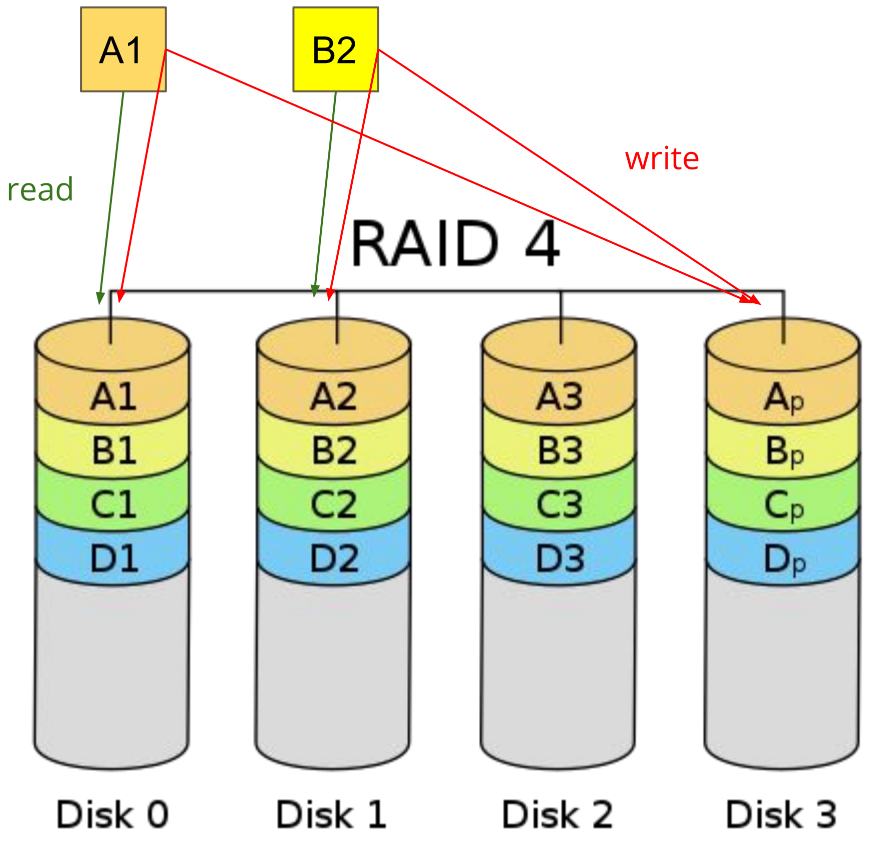


Figure from Wikipedia

RAID 5

- Block striping
- Distributed parity

$$Ap = A1 \oplus A2 \oplus A3$$

$$A1 = Ap \oplus A2 \oplus A3$$

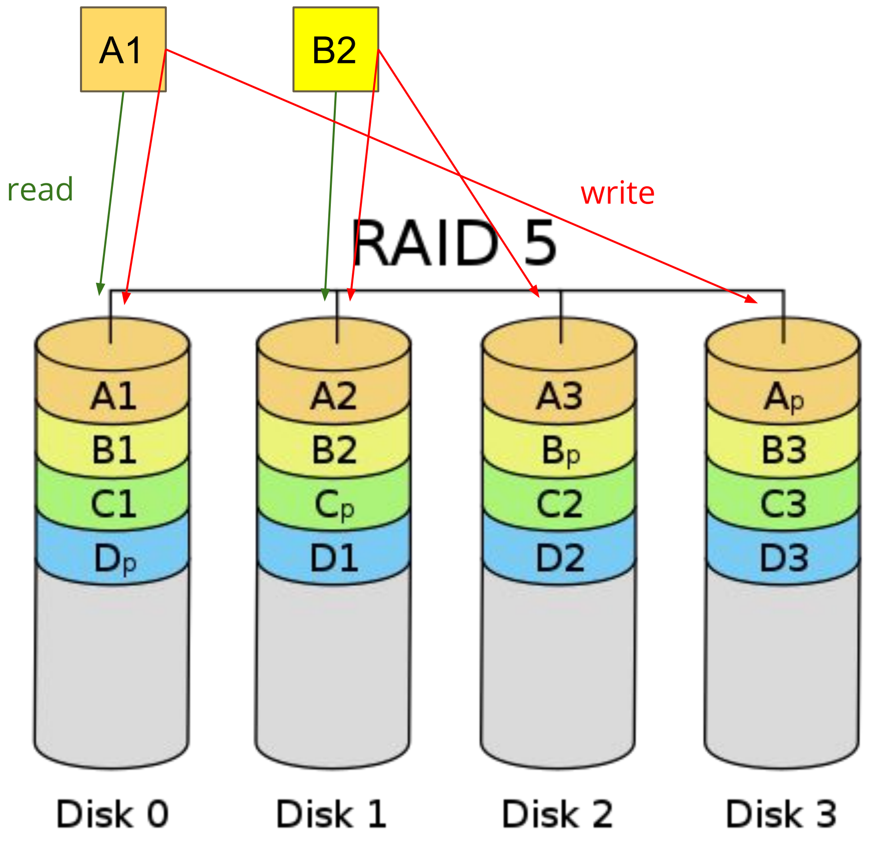


Figure from Wikipedia

Disk failure recovery

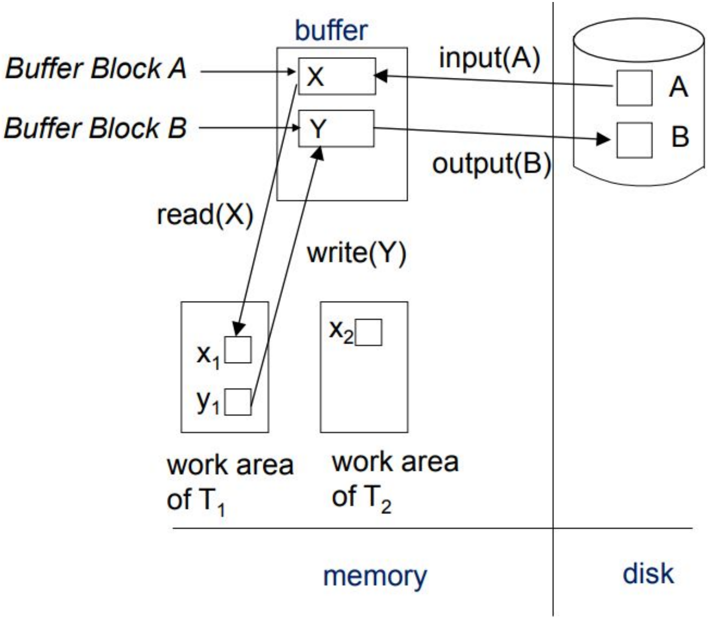
- One disk failure
  - If RAID 4 or above, recover data
  - Sector failures are subsumed
- Parity can help recover full disk failure
  - Identify disk failure
  - compute parity with remaining disks
  - Write to a replacement disk
- What if multiple disks fail?

System crash

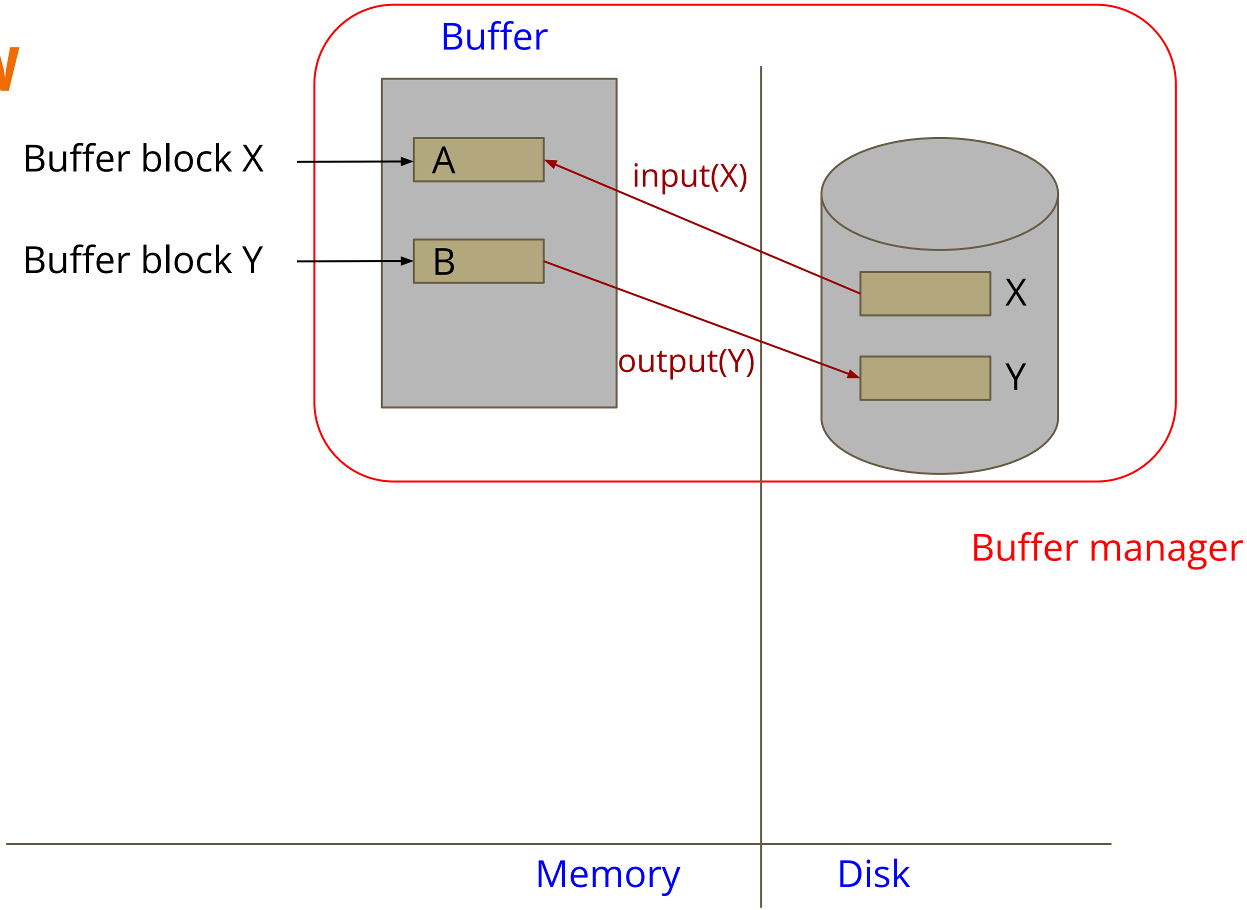
- Volatile storage lost
  - Main memory
  - Cache
- Non-volatile storage survives
  - Disks
  - Flash memory
  - NVRAM
- Assumptions
  - Disk does not fail at the same time
  - Reasonable?

System crash

- Buffer manager and Transaction manager are independent

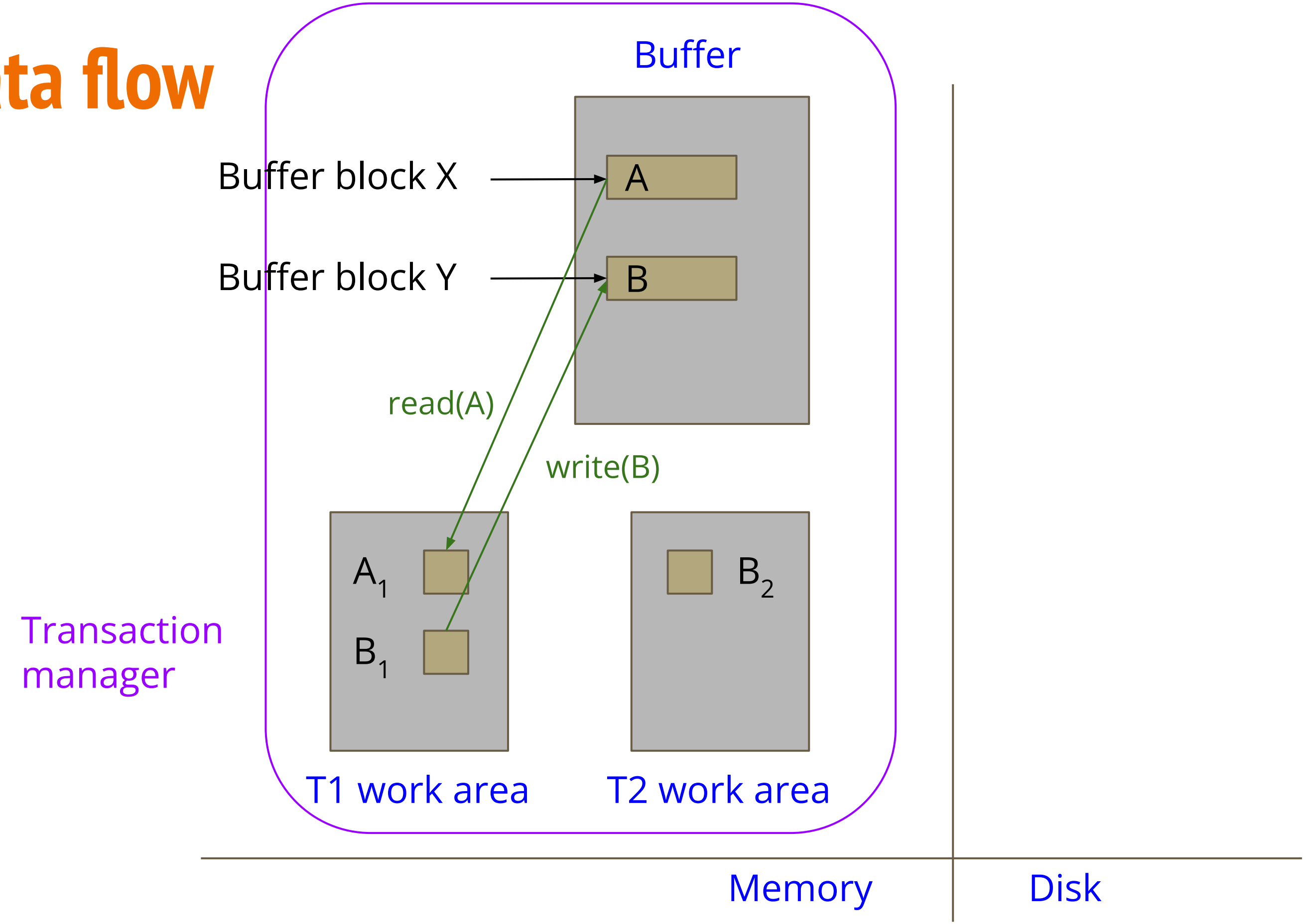


Data flow



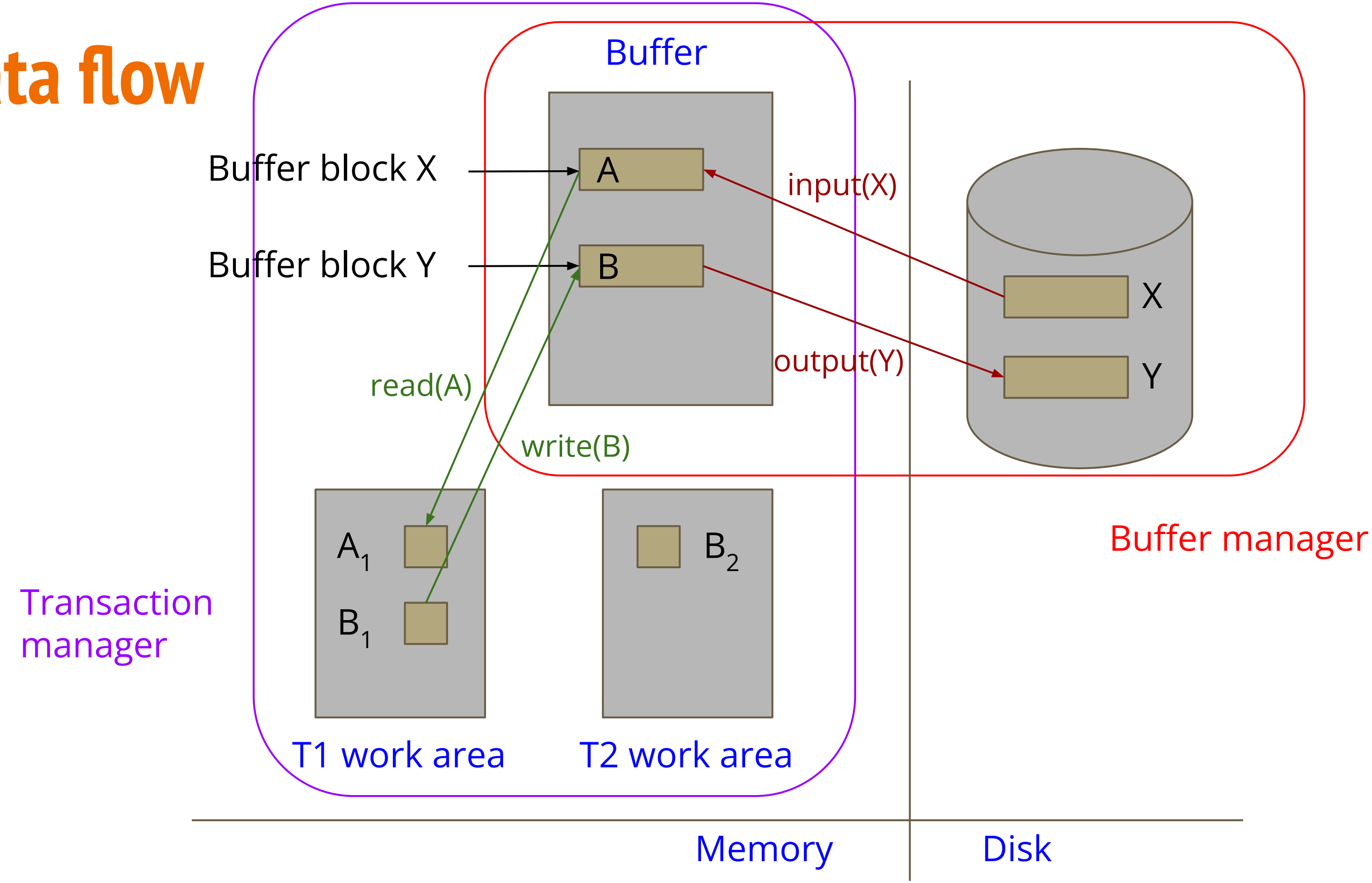
Buffer
Buffer block X
Buffer block Y
A
B
input(X)
output(Y)
X
Y
Buffer manager
Memory
Disk

Data flow



Buffer
Buffer block X
A
Buffer block Y
B
read(A)
write(B)
A1
B1
B2
Transaction manager
T1 work area
T2 work area
Memory
Disk

Data flow


Buffer
Buffer block X
A
input(X)
Buffer block Y
B
X
output(Y)
Y
read(A)
write(B)
Buffer manager
A1
B2
B1
Transaction manager
T1 work area
T2 work area
Memory
Disk

System crash

- Buffer manager and Transaction manager are independent
  - Output to disk any time after write to buffer
  - Before or after transaction commits
- Point of failure
  - After T1 commits but contents not updated in disk
  - After B is written to disk but A is not
  - T1 not committed but B is written to disk

Transaction T1
read(A)
A = A - 50
write(A)
read(B)
B = B + 50
write(B)
commit
crash
output(B)
output(A)

System crash

- Buffer manager and Transaction manager are independent
    - Output to disk any time after write to buffer
    - Before or after transaction commits

- Point of failure
    - After T1 commits but contents not updated in disk
    - After B is written to disk but A is not
    - T1 not committed but B is written to disk

Transaction T1
read(A)
A = A - 50
write(A)
read(B)
B = B + 50
write(B)
commit
output(B)
crash
output(A)

System crash

- Buffer manager and Transaction manager are independent
  - Output to disk any time after write to buffer
  - Before or after transaction commits

- Point of failure
  - After T1 commits but contents not updated in disk
  - After B is written to disk but A is not
  - T1 not committed but B is written to disk

Transaction T1
read(A)
A = A - 50
write(A)
read(B)
B = B + 50
write(B)
output(B)
crash
commit
output(A)

Recovery algorithms

- Steps during execution
  - Write enough data to disk
- Steps during recovery
  - Use the written data to recover
- End goal
  - Write least and still recover fast

Simple recovery

- Enforce restrictions on buffer manager and transaction manager
    - Force writes to disk
    - Uncommitted writes over committed data disallowed

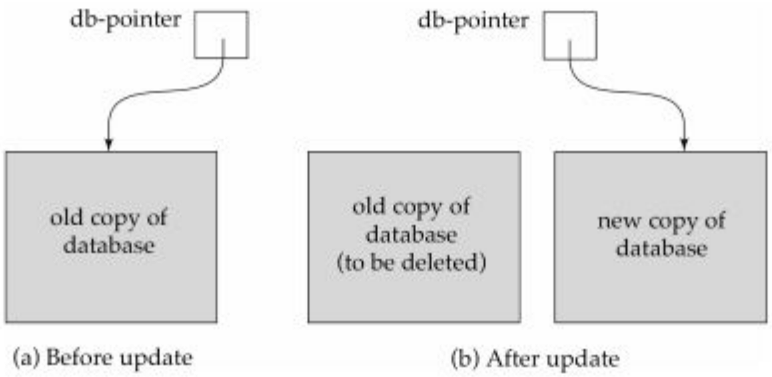
Drawbacks?

Simple recovery

- Shadow copy
  - Pointer to database in disk

- Shadow paging

Drawbacks?

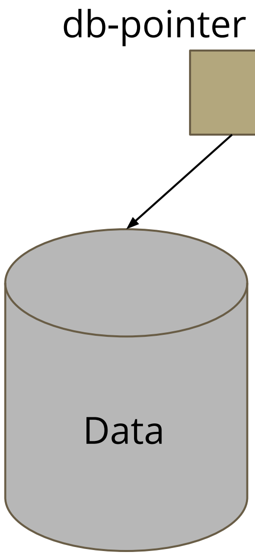


Simple recovery

- Shadow copy
    - Pointer to database in disk

- Shadow paging

Drawbacks?

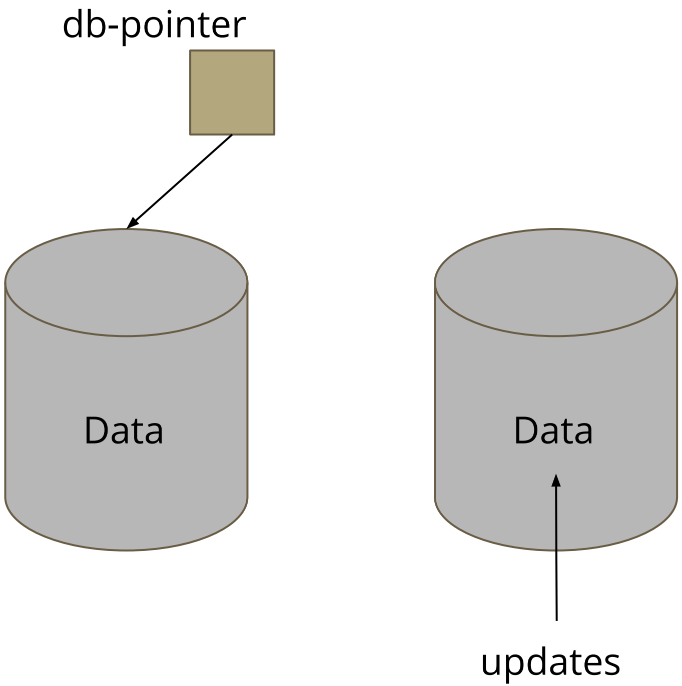


Simple recovery

- Shadow copy
  - Pointer to database in disk

- Shadow paging

Drawbacks?

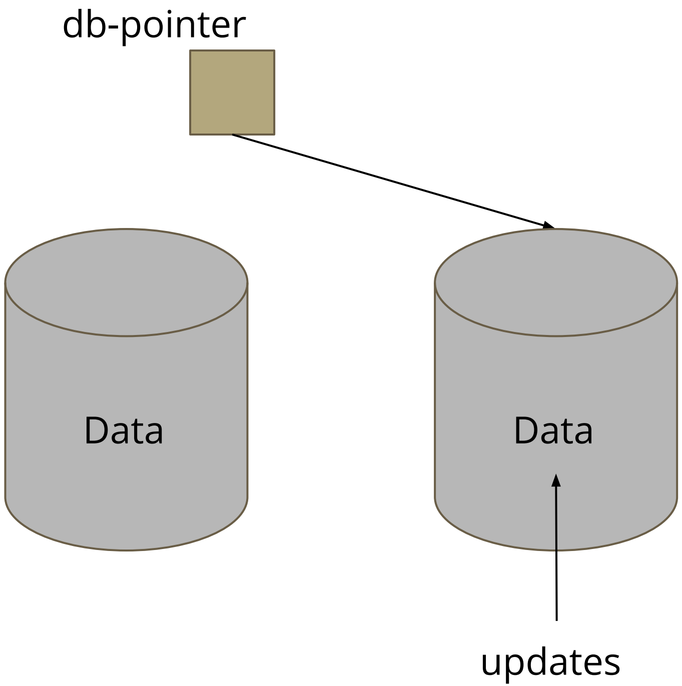


Simple recovery

- Shadow copy
    - Pointer to database in disk

- Shadow paging

Drawbacks?



Logging

- Record what transactions do
- Only delta/update to save space and time
- Ordered list
  - Sequential writes
- Assume to stable storage for now

- What to log?
  - T1 starts
  - write(A)
  - write(B)
  - T1 commits
  - T1 aborts

Transaction T1
read(A)
A = A - 50
write(A)
read(B)
B = B + 50
write(B)
commit

Redo

- Why we need redo?
  - Durability guarantee
  - For committed transactions
  - Redo their actions
- Write operation logging
  - Transaction identifier
  - Data item identifier
  - Value that was written

Transaction T1
read(A)
A = A - 50
write(A)
read(B)
B = B + 50
write(B)
commit

<T1 start>
<T1, A, 950>
<T1, B, 2050>
<T1 commit>

Undo

- Why we need undo?
  - Atomicity guarantee
  - For uncommitted transactions
  - Undo their actions
- Write operation logging
  - Transaction identifier
  - Data item identifier
  - Old value that was overwritten
  - New value that was written

Transaction T1
read(A)
A = A - 50
write(A)
read(B)
B = B + 50
write(B)
commit

<T1 start>
<T1, A, 1000, 950>
<T1, B, 2000, 2050>
<T1 commit>

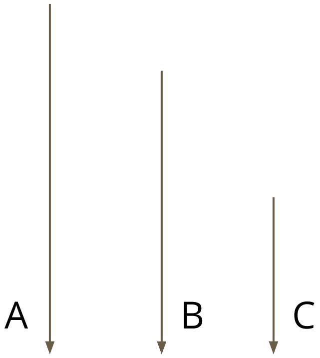
Crash recovery

Starting values {A:1000, B:2000, C:700}

Transaction T1
read(A)
A = A - 50
write(A)
read(B)
B = B + 50
write(B)
commit

Transaction T2
read(C)
C = C - 100
write(C)
commit

Possible log
<T1 start>
<T1, A, 1000, 950>
<T1, B, 2000, 2050>
<T1 commit>
<T2 start>
<T2, C, 700, 600>
<T2 commit>



Crash recovery

Possible logs after crash

Possible log
<T1 start>
<T1, A, 1000, 950>
<T1, B, 2000, 2050>

crash

T1 incomplete (undo)

Possible log
<T1 start>
<T1, A, 1000, 950>
<T1, B, 2000, 2050>
<T1 commit>
<T2 start>
<T2, C, 700, 600>

crash

T1 complete (redo)
T2 incomplete (undo)

Possible log
<T1 start>
<T1, A, 1000, 950>
<T1, B, 2000, 2050>
<T1 commit>
<T2 start>
<T2, C, 700, 600>
<T2 commit>

crash

T1 complete (redo)
T2 complete (redo)

Write ahead logging (WAL)

- Logs are buffered too
  - Before written out to disk
  - Can be lost during crash
  - What needs to change?

- Transactions commit only when their commit log is written to disk
  - All update logs of transaction are written to disk before the commit log
- Write ahead logging
  - Write logs of a data item first than its data block
  - How to enforce this?

Checkpoint

- Savepoint at which everything is safe in disk

- Why?
  - Need to read the whole log on failure
  - Log can grow too large
  - Larger log means more time to recover
  - Older redos are likely not needed

Checkpoint

- Write all logs to disk
- Write all data to disk
- Append to disk log <checkpoint, L>

- No updates allowed during checkpoint

Recovery with checkpoint

- If commit appears before checkpoint
    - Nothing to be done
- If transaction is in L
    - Redo if committed before crash
    - Undo if uncommitted
- If a transaction started after checkpoint
    - Redo if committed before crash
    - Undo if uncommitted

ARIES

- ARIES - Algorithms for Recovery and Isolation Exploiting Semantics
  - Current state of the art in recovery
  - Developed in 90s at IBM
  - Used in many databases (flavours of ARIES)

- How is it different from what we have seen so far!!

ARIES

- Main features
    - Write ahead logging
    - Repeating history during redo
    - Logging changes during undo

- Main advantages
    - Reduced recovery time
    - Reduce checkpoint overhead
    - Supports physiological redo operations
    - Supports transaction savepoints
    - Supports fine grained locking

Log sequence number

- Log sequence number (LSN)
    - Identify log records uniquely
    - Always increasing
- flushedLSN
    - Last log record written to disk
- pageLSN
    - Maintained for each page
    - LSN of the log record that last modified the page
- prevLSN
    - For each log record
    - Last log record by same transaction

Log record type

- Type of log record
  - For each log record
  - Store only required information
- Update log records
  - Page id, length and offset
  - Instead of data item identifier
- End log record
  - After commit or abort, some bookkeeping needed
  - End log record to denote that

prevLSN	Transaction id	type	More fields based on type

Compensation log record

- Undo operations
  - New write that changes value
  - Add to log
  - Doesn’t need undo information
  - undoNextLSN, LSN of next record to be undone
  - Why needed?

- Redo operations
  - Not required, why?

T1	Update log
T2	Update log
T2	Update log
T1	Update log
T1	Abort log
T1	CLR

State information

- In-memory tables
- Transaction table
  - One entry per active transaction
  - lastLSN - most recent log record
  - Maintained by transaction manager
- Dirty page table
  - One entry per dirty page in buffer
  - recLSN - first LSN that made it dirty
  - Maintained by buffer manager
- Recreated during recovery

ARIES optimizations

- Skip unnecessary redo operations
- Skip already undone undo records
- Fuzzy checkpoint
  - Begin and end checkpoint log
  - Transaction table and dirty page table from begin time
  - Account for log records in between during recovery

ARIES crash recovery

- Analysis phase
  - Determine redo starting record
  - Determine active transactions at crash
  - Determine possible dirty pages at crash
- Redo phase
  - Redo all updates
  - Redo action and update pageLSN
- Undo phase
  - Undo uncommitted transactions
  - Reverse chronological order
  - Undo action and write CLR

Analysis phase

1. Restore state tables from last checkpoint
2. Scan forward and update tables
    a. Remove transaction if end record found
    b. Update lastLSN and status otherwise
    c. Add pages to dirty page table for update logs

At the end,

- Transaction table has active transaction list during crash
- Dirty page table has all possible dirty pages during crash

Redo phase

1. Scan forward from smallest recLSN in dirty page table
2. Reapply all updates, unless
    a. Affected page is not in dirty page table
    b. recLSN of the page > redo record LSN
    c. pageLSN of the page >= redo record LSN

At the end,

- Crash situation recreated by repeating history

Undo phase

1. Get all active transactions during crash (analysis phase)
2. toUndo is a set of lastLSN of these transactions
3. Repeat, until toUndo is empty
    a. Choose and remove largest LSN from toUndo
    b. If LSN is CLR and undoNextLSN is null, write an end log record
    c. If LSN is CLR and undoNextLSN is not null, add undoNextLSN to toUndo
    d. If LSN is update, undo the update, write CLR, add prevLSN to toUndo

At the end, recovery is complete. Normal operation can resume.

Crash during recovery

- During analysis phase
  - All information lost
  - Begin again with same information after restart
- During redo phase
  - Some pages might have been written to disk after redo
  - After restart, we get updated pageLSN for those pages and redo can be skipped
- During undo phase
  - Some undo operations might have been performed with added CLR log record
  - Some pages might have been written to disk
  - After restart, the CLRs will be redone in redo phase
  - Undo proceeds with the undoNextLSN of the last CLR

Transaction failure

- No crash occurred
- But transaction failed

- Special case of undo phase
- Read log backwards
- Perform undo for failed transaction
- Write CLRs

Data center failure

- Multiple data centers/nodes
- Distributed database
  - Master slave model

- Data duplication
  - Copy data from master to slave periodically
- Log replication
  - Replicate log to slave from master