

Recovery

Lecture 16



Outline

1. Issues and Models

- Transaction Properties
- Storage Hierarchy
- Failure Mode
- System Log

2. UNDO Logging

- (Quiescent) Checkpoints

3. REDO Logging

4. UNDO/REDO Logging

5. Closing Notes



Transaction Properties

This lecture focuses on how to support transactions in the context of certain types of failures

- **Atomicity:** need to UNDO writes from partially completed transactions
- **Consistency:** need to make sure that after failure, new transactions have a consistent start state
- **Durability:** need to REDO writes from committed transactions



Reminder: Consistency

- Most of the time we think of consistency from the DBMS standpoint
 - Often in context of failure, concurrency
- But it may be the case that transactions themselves are poorly written w.r.t. database constraints
 - And thus are legitimately aborted



Checkup

- Assume a database has the following asserted constraint: $A > B > 0$
 - Which transactions will NOT necessarily preserve consistency of the database?
 - Provide an example
-
- i. $A = 2A$; $B = 2B$
 - ii. $A = 2A$; $B = A - 1$



Answer ($A > B > 0$)

i. $A = 2A; B = 2B$

- WILL preserve
- If both started > 0 , will remain so under multiplication
- If $A > B$, $2A > 2B$

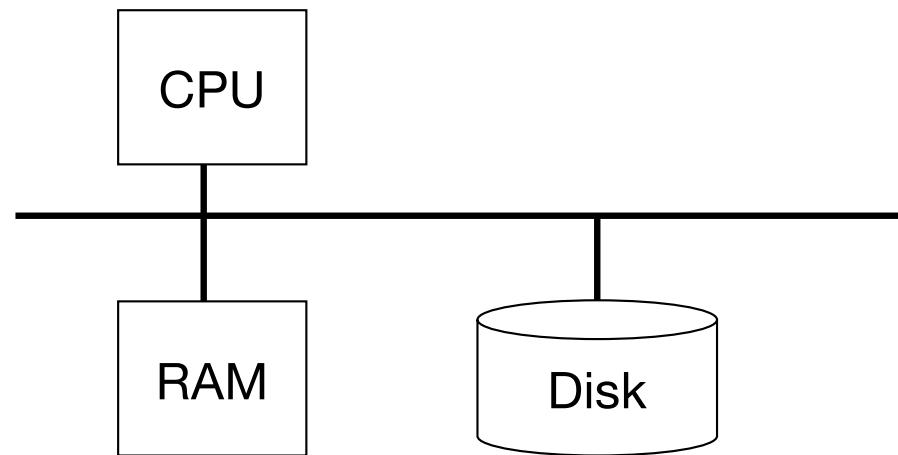
ii. $A = 2A; B = A - 1$

- WILL NOT (always) preserve
- Start: $A=0.5, B=0.4$
 - Result: $A=1, B=0$



Storage Hierarchy

Assume the following computational abstraction...



where RAM is volatile (contents do not durably persist unless directed by CPU to disk)



Operational Sequence

1. Input(X): Disk -> Memory

2. Read(X, t): Memory -> Variable

- Necessary for any computation via CPU
- Assumed to perform Input(X) if necessary

3. Write(X, t): Variable -> Memory

4. Output(X): Memory -> Disk



Failure Modes

- User error (e.g., bad data entry)
 - Some handled via (application) constraints
 - Not discussed here
- Media failure (e.g., bad HD/SSD)
 - Briefly touched on at the end
- Catastrophic (e.g., fire, flood)
 - Briefly touched on at the end
- System failure (e.g., OS/application crash)
 - What happens when RAM doesn't make it to disk?
 - **Our focus**



System Log

- AKA transaction log, recovery log, ...
- Think of it as an append-only file on disk
 - Known format (sequence of “records”)
 - Maintained by the DBMS
 - As with actual DB data, buffered in RAM to improve efficiency (but only disk contents survive failure)
- **Main question:** what set of rules can we put in place to ensure correct transaction processing in the face of system failure, while not severely impacting performance?

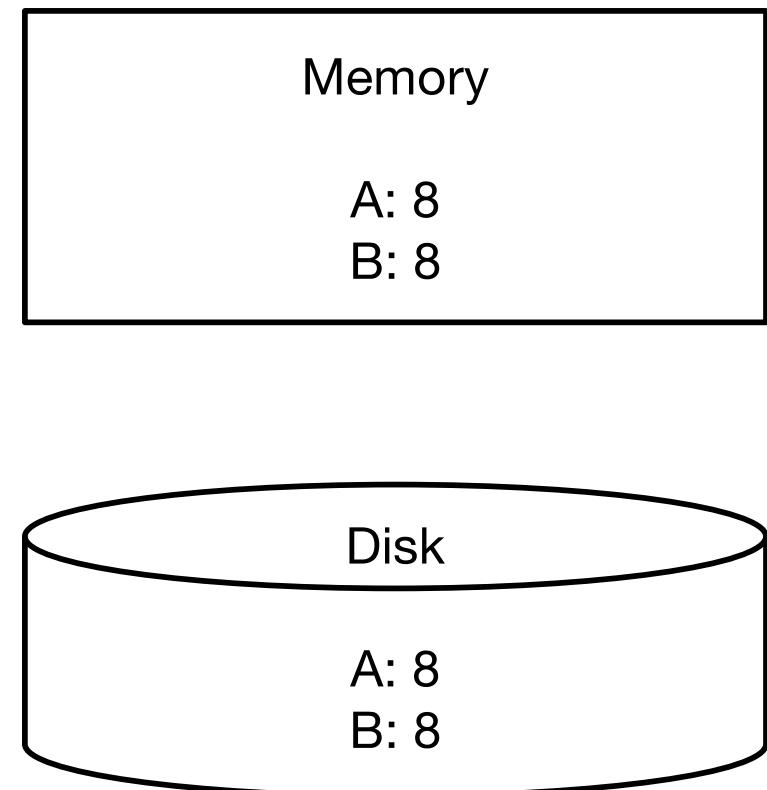


Example

- Constraint: A=B
- T1: A=2A; B=2B
 - Transaction consistent?

T1

1. Read(A, t)
2. $t = 2*t$
3. Write(A, t)
4. Read(B, t)
5. $t = 2*t$
6. Write(B, t)
7. Output(A)
8. Output(B)

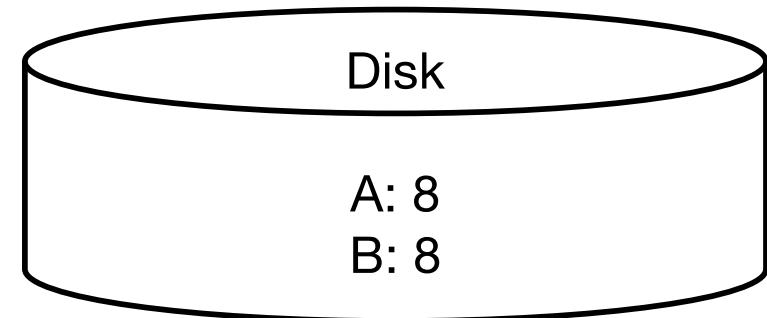
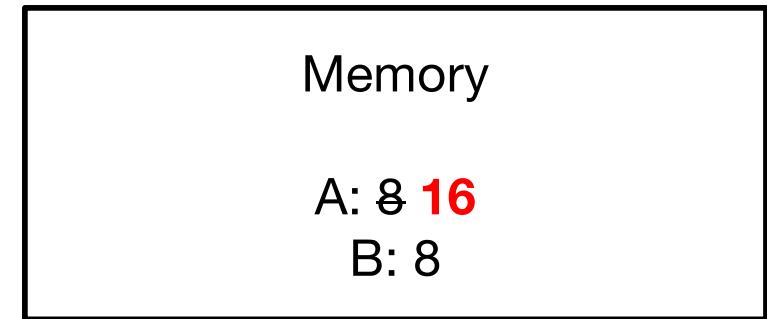


Example

- Constraint: A=B
- T1: A=2A; B=2B
 - Transaction consistent?

T1

1. Read(A, t)
2. $t = 2^*t$
- 3. Write(A, t)**
4. Read(B, t)
5. $t = 2^*t$
6. Write(B, t)
7. Output(A)
8. Output(B)

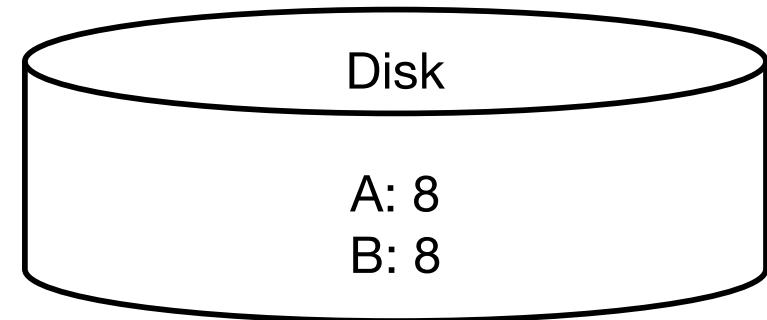
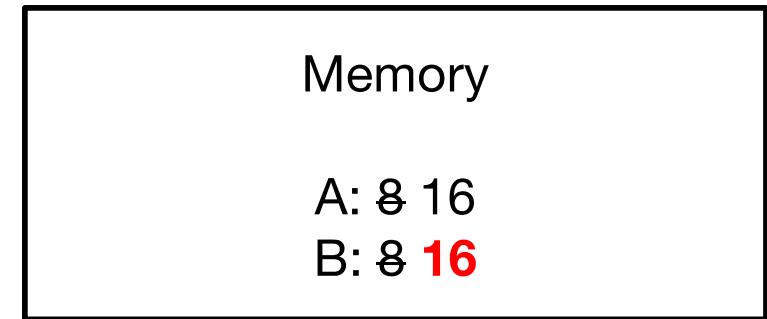


Example

- Constraint: A=B
- T1: A=2A; B=2B
 - Transaction consistent?

T1

1. Read(A, t)
2. $t = 2^*t$
3. Write(A, t)
4. Read(B, t)
5. $t = 2^*t$
- 6. Write(B, t)**
7. Output(A)
8. Output(B)



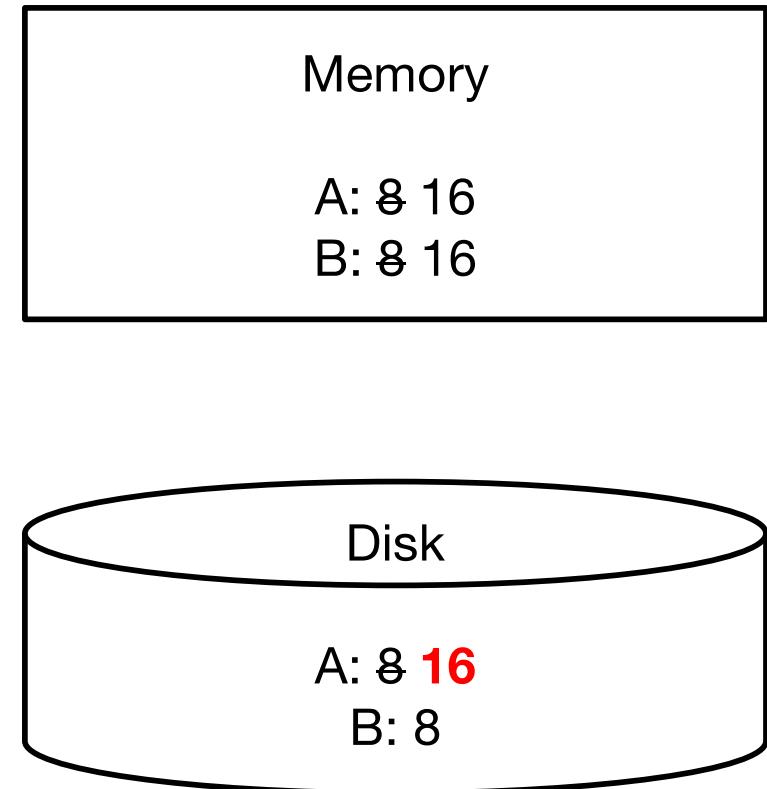
Recovery

Example

- Constraint: A=B
- T1: A=2A; B=2B
 - Transaction consistent?

T1

1. Read(A, t)
2. $t = 2*t$
3. Write(A, t)
4. Read(B, t)
5. $t = 2*t$
6. Write(B, t)
7. **Output(A)**
8. Output(B)

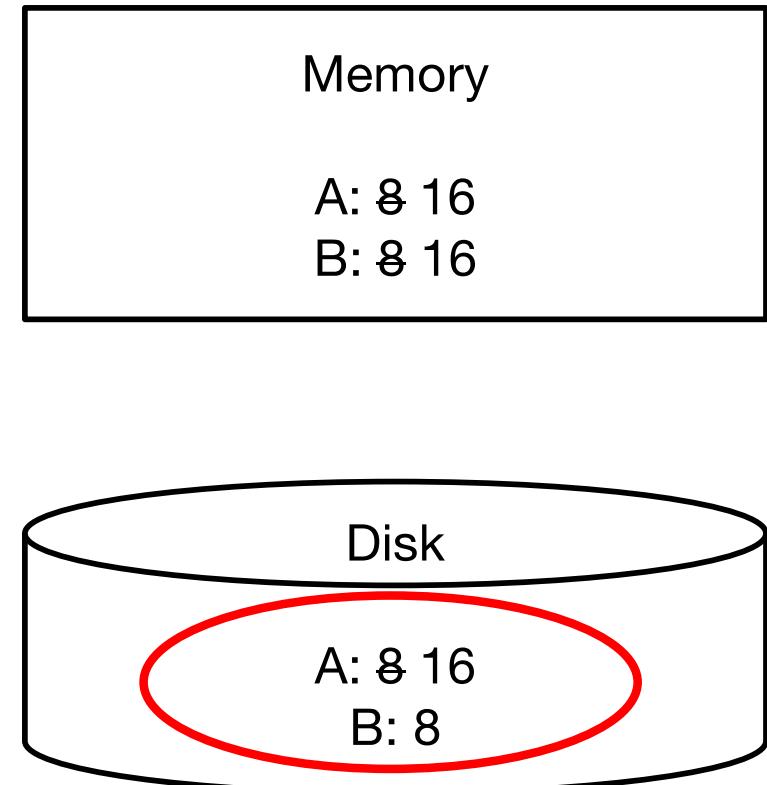


Example

- Constraint: $A=B$
- T1: $A=2A$; $B=2B$
 - Transaction consistent?

T1

1. Read(A, t)
 2. $t = 2*t$
 3. Write(A, t)
 4. Read(B, t)
 5. $t = 2*t$
 6. Write(B, t)
 7. Output(A)
-
8. Output(B)



Consistency Fail – Now What!!??



Recovery

Map from Here

Log Format

- Approach #1: UNDO
 - Write data before COMMIT
- Approach #2: REDO
 - All modifications in RAM until log on disk
- Approach #3: UNDO/REDO
 - Fat log records

Long-Term Strategy

- Approach #1: None
 - Good luck with that...
- Approach #2: Checkpoint
 - Stops all transactions
- Approach #3: Non-Quiescent Checkpoint
 - Need more history



UNDO Logging

- At the beginning of a transaction
 $\langle T_i, \text{ START} \rangle$
- For every action: generate log record containing old value
 $\langle T_i, X, v \rangle$
- **Before** X is modified on disk (via output), log records pertaining to X must be on disk
 - Write-Ahead Logging (WAL)
 - FLUSH LOG
- **Before** commit is flushed to log, all writes of transaction must be reflected on disk
 $\langle T_i, \text{ COMMIT} \rangle$



Example

Step	Action	t	M-A	M-B	D-A	D-B	Log
------	--------	---	-----	-----	-----	-----	-----

- $M-*$ = Value * in memory
- $D-*$ = Value * on disk



Example



Example



Example



Example



Example



Example

Step	Action	t	M-A	M-B	D-A	D-B	Log
1					8	8	<T1, START>
2	READ(A, t)	8	8		8	8	
3	$t=2*t$	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T1, A, 8>
5	READ(B, t)	8	16	8	8	8	
6	$t=2*t$	16	16	8	8	8	



Example

Step	Action	t	M-A	M-B	D-A	D-B	Log
1					8	8	<T1, START>
2	READ(A, t)	8	8		8	8	
3	t=2*t	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T1, A, 8>
5	READ(B, t)	8	16	8	8	8	
6	t=2*t	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T1, B, 8>



Example

Step	Action	t	M-A	M-B	D-A	D-B	Log
1					8	8	<T1, START>
2	READ(A, t)	8	8		8	8	
3	t=2*t	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T1, A, 8>
5	READ(B, t)	8	16	8	8	8	
6	t=2*t	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T1, B, 8>
8	FLUSH LOG	16	16	16	8	8	



Example

Step	Action	t	M-A	M-B	D-A	D-B	Log
1					8	8	<T1, START>
2	READ(A, t)	8	8		8	8	
3	t=2*t	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T1, A, 8>
5	READ(B, t)	8	16	8	8	8	
6	t=2*t	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T1, B, 8>
8	FLUSH LOG	16	16	16	8	8	
9	OUTPUT(A)	16	16	16	16	8	



Example

Step	Action	t	M-A	M-B	D-A	D-B	Log
1					8	8	<T1, START>
2	READ(A, t)	8	8		8	8	
3	t=2*t	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T1, A, 8>
5	READ(B, t)	8	16	8	8	8	
6	t=2*t	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T1, B, 8>
8	FLUSH LOG	16	16	16	8	8	
9	OUTPUT(A)	16	16	16	16	8	
10	OUTPUT(B)	16	16	16	16	16	



Example

Step	Action	t	M-A	M-B	D-A	D-B	Log
1					8	8	<T1, START>
2	READ(A, t)	8	8		8	8	
3	t=2*t	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T1, A, 8>
5	READ(B, t)	8	16	8	8	8	
6	t=2*t	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T1, B, 8>
8	FLUSH LOG	16	16	16	8	8	
9	OUTPUT(A)	16	16	16	16	8	
10	OUTPUT(B)	16	16	16	16	16	
11		16	16	16	16	16	<T1, COMMIT>



Example

Step	Action	t	M-A	M-B	D-A	D-B	Log
1					8	8	<T1, START>
2	READ(A, t)	8	8		8	8	
3	t=2*t	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T1, A, 8>
5	READ(B, t)	8	16	8	8	8	
6	t=2*t	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T1, B, 8>
8	FLUSH LOG	16	16	16	8	8	
9	OUTPUT(A)	16	16	16	16	8	
10	OUTPUT(B)	16	16	16	16	16	
11		16	16	16	16	16	<T1, COMMIT>
12	FLUSH LOG	16	16	16	16	16	



UNDO Recovery

- For all transactions that did not COMMIT/ROLLBACK...
 - Remember, if these log entries made it to disk, so did the corresponding value operations
- From **latest to earliest** of $\langle T_i, X, v \rangle$
 - WRITE(X, v)
 - OUTPUT(X)
- Write $\langle T_i, ABORT \rangle$
 - Note, system failure during recovery doesn't hurt, because operations **idempotent**



Example Failure (UNDO)

Step	Action	t	M-A	M-B	D-A	D-B	Log
							<T1, START>
							<T1, A, 8>
							<T1, B, 8>
							<T1, COMMIT>



Example Failure (UNDO)

Step	Action	t	M-A	M-B	D-A	D-B	Log
1					8	8	<T1, START>
2	READ(A, t)	8	8		8	8	
3	t=2*t	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T1, A, 8>
5	READ(B, t)	8	16	8	8	8	
6	t=2*t	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T1, B, 8>
8	FLUSH LOG	16	16	16	8	8	
9	OUTPUT(A)	16	16	16	16	8	
10	OUTPUT(B)	16	16	16	16	16	
11		16	16	16	16	16	<T1, COMMIT>
12	FLUSH LOG	16	16	16	16	16	

Do nothing!
Commit on
disk!



Example Failure (UNDO)

Step	Action	t	M-A	M-B	D-A	D-B	Log
							<T1, START>
							<T1, A, 8>
							<T1, B, 8>



Example Failure (UNDO)

Step	Action	t	M-A	M-B	D-A	D-B	Log
1					8	8	<T1, START>
2	READ(A, t)	8	8		8	8	
3	$t=2*t$	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T1, A, 8>
5	READ(B, t)	8	16	8	8	8	
6	$t=2*t$	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T1, B, 8>
8	FLUSH LOG	16	16	16	8	8	
9	OUTPUT(A)	16	16	16	16	8	
10	OUTPUT(B)	16	16	16	16	16	

UNDO all T1!
Unsure if any
OUTPUTS
finished :(



Checkup

Given the following UNDO log, what **could** be the values of A and B on disk?

- R: A=A+1; B=B-1
- S: A=2*A; B=B+1
- T: A=A+1; B=3*B

1. <S, START>
2. <T, START>
3. <S, A, 5>
4. <S, B, 10>
5. <S, COMMIT>
6. <T, A, 10>
7. <T, B, 11>
8. <R, START>
9. <T, COMMIT>
10. <R, A, 11>
11. <R, B, 33>



Answer

Given the following UNDO log, what **could** be the values of A and B on disk?

- R: A=A+1; B=B-1
- S: A=2*A; B=B+1
- T: A=A+1; B=3*B

1. <S, START>
2. <T, START>
3. <S, A, 5>
4. <S, B, 10>
5. <S, COMMIT>
6. <T, A, 10>
7. <T, B, 11>
8. <R, START>
9. <T, COMMIT>
10. <R, A, 11>
11. <R, B, 33>

COMMIT made it to disk for S & T, so their modifications certainly made it to disk.

Last known values (@9): A=11, B=33

COMMIT didn't make it to disk for R, so unsure if any changes made it to disk. Also aren't unsure of OUTPUT order in R, so...

A=11, B=33; A=12, B=32

A=12, B=33; A=11, B=32



In the Long Run...

- Even if average number of records remains the same, the log will grow monotonically
 - As will the time to recover!
- Hence the need for checkpointing
 - Identify a point past which log is unnecessary (for recovery)



Basic Checkpoint

1. Stop accepting new transactions
2. Wait until all active transactions COMMIT or ROLLBACK (in the log!)
 - Termed: **quiescence**
3. FLUSH the log
4. Write <CKPT> to the log, FLUSH
 - Point to stop checking for unfinished transactions in future recovery – hence, could delete before!
5. Resume accepting transactions



Nonquiescent Checkpoint

- Step 1 (stop all incoming transactions) might not be reasonable in all work loads
 - Basically takes the database offline :(
- SO...
 1. Write **<START CKPT(T1, ... Tk)>**
 - Includes transactions that have started, but not yet completed
 2. When T1...Tk have completed, **<END CKPT>** and flush the log



Recovery

- What happens if we hit an END CKPT?
 - Scan back only as far as START: any incomplete started after that point
- What happens if we hit a CKPT START?
 - Failure must have happened during checkpointing :(
 - BUT, we know exactly which incomplete transactions ($T_1 \dots T_k$), so go back as far as their START
- Win: we only need to go back as far as START CKPT since the last END CKPT



REDO Logging

- A weakness of UNDO: need to write to disk before COMMIT
- REDO protocol (via **Deferred Update**)
 - For every action, generate redo log record (containing new value)
 - Before X is modified on disk, all log records for the transaction that modified X (including COMMIT) must be on disk



REDO Recovery

- For all transactions that did COMMIT...
 - Remember, if these log entries made it to disk, the whole transaction committed
- From **earliest to latest** of $\langle T_i, X, v \rangle$
 - $\text{WRITE}(X, v)$
 - $\text{OUTPUT}(X)$
 - Technically, optional (why?)



REDO Checkpoints

1. START(...), FLUSH LOG
 2. Write DB changes for committed transactions
 3. END, FLUSH LOG
-
- Note: requires that the DBMS keep track of changes to memory by committed transactions that have not yet been written to disk



Checkup

Assume that initially A=5, B=10. Given the following REDO log, what **could** be the values of A and B on disk?

1. <S, START>
2. <S, A, 20>
3. <S, B, 30>
4. <S, COMMIT>
5. <R, START>
6. <R, B, 25>
7. <T, START>
8. <R, A, 50>
9. <T, B, 2>
10. <T, COMMIT>



Answer

Assume that initially A=5, B=10. Given the following REDO log, what **could** be the values of A and B on disk?

1. <S, START>
2. <S, A, 20>
3. <S, B, 30>
4. <S, COMMIT>
5. <R, START>
6. <R, B, 25>
7. <T, START>
8. <R, A, 50>
9. <T, B, 2>
10. <T, COMMIT>

COMMIT made it to disk for S & T, so their modifications *could* have made it to disk.

COMMIT for R did not make it to disk, so new values certainly could not.

So... $\{5, 20\} \times \{10, 30, 2\}$

A=5, B=10; A=5, B=30; A=5, B=2
A=20, B=10; A=20, B=30; A=20, B=2



UNDO vs REDO

- UNDO: write all before COMMIT
 - High I/O :(
- REDO: in RAM till COMMIT
 - Higher memory footprint, greater need for recovery in case of failure :(
- Solution: combine as UNDO/REDO!



UNDO/REDO Logging

- Before X is modified on disk, flush corresponding log record
 $\langle T_i, X, \text{old}, \text{new} \rangle$
- Flush log on commit
- Full flexibility as to when non-log data is written to disk
 - At the cost of fatter log entries



Recovery

- Backwards pass (UNDO)
 - Start at the end of the log, go backwards to the most recent checkpoint start
 - UNDO actions of uncommitted transactions
 - Including ABORT entry
- Forward pass (REDO)
 - Start at the most recent checkpoint start, go forward to the end of the log
 - REDO actions of committed transactions



UNDO/REDO Checkpoints

1. START(...), FLUSH LOG
2. Write all DB changes (committed or not)
3. END, FLUSH LOG



Checkup

Given the following UNDO/REDO log, what **could** be the values of A and B on disk?

1. <S, START>
2. <S, A, 10, 20>
3. <S, B, 15, 30>
4. <S, COMMIT>
5. <T, START>
6. <T, A, 20, 30>
7. <START CKPT(T)>
8. <T, B, 30, 10>
9. <T, COMMIT>
10. <R, START>
11. <R, A, 30, 50>
12. <END CKPT>
13. <R, B, 10, 1>
14. <R, COMMIT>



Answer

Given the following UNDO/REDO log, what **could** be the values of A and B on disk?

1. <S, START>
2. <S, A, 10, 20>
3. <S, B, 15, 30>
4. <S, COMMIT>
5. <T, START>
6. <T, A, 20, 30>
7. <START CKPT(T)>
8. <T, B, 30, 10>
9. <T, COMMIT>
10. <R, START>
11. <R, A, 30, 50>
12. <END CKPT>
13. <R, B, 10, 1>
14. <R, COMMIT>

At the end of the checkpoint, last memory values were on disk (A=30, B=30).

Any other log entries *could* have been mirrored to disk by a delayed update.

SO... $\{30, 50\} \times \{30, 10, 1\}$

A=30, B=30; A=30, B=10; A=30, B=1
A=50, B=30; A=50, B=10; A=50, B=1



Checkup

Given the following UNDO/REDO log, what are the values of A/B **in the database at the end of recovery?**

1. <S, START>
2. <S, A, 10, 20>
3. <S, B, 15, 30>
4. <S, COMMIT>
5. <T, START>
6. <T, A, 20, 30>
7. <START CKPT(T)>
8. <T, B, 30, 10>
9. <T, COMMIT>
10. <R, START>
11. <R, A, 30, 50>
12. <END CKPT>
13. <R, B, 10, 1>
14. <START CKPT(P, R)>
15. <P, START>
16. <R, COMMIT>
17. <P, A, 50, 60>



Answer

Given the following UNDO/REDO log, what are the values of A/B **in the database at the end of recovery?**

1. <S, START>
2. <S, A, 10, 20>
3. <S, B, 15, 30>
4. <S, COMMIT>
5. <T, START>
6. <T, A, 20, 30>
7. <START CKPT(T)>
8. <T, B, 30, 10>
9. <T, COMMIT>
10. <R, START>
11. <R, A, 30, 50>
12. <END CKPT>
13. <R, B, 10, 1>
14. <START CKPT(P, R)>
15. <P, START>
16. <R, COMMIT>
17. <P, A, 50, 60>

All transactions but P have committed via the log, so look to last result values...

A=50

B=1



Closing Notes

- Still need database/log copies to protect against media failure
 - RAID, remote backup (full vs incremental)
- The log will likely grow faster than the db
 - So typical to regularly delete (“truncate”) unnecessary entries (of course, when to do so depends on logging type)

