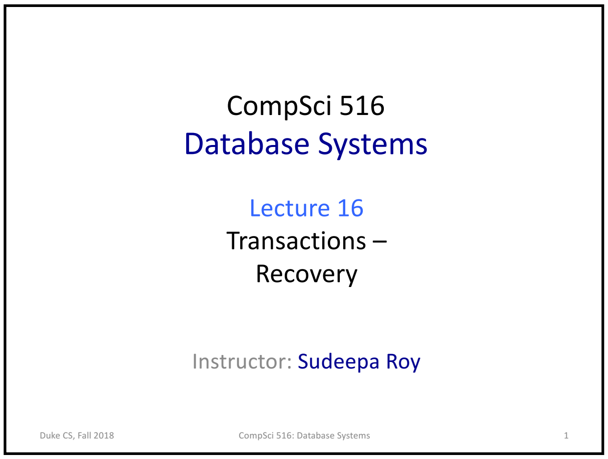


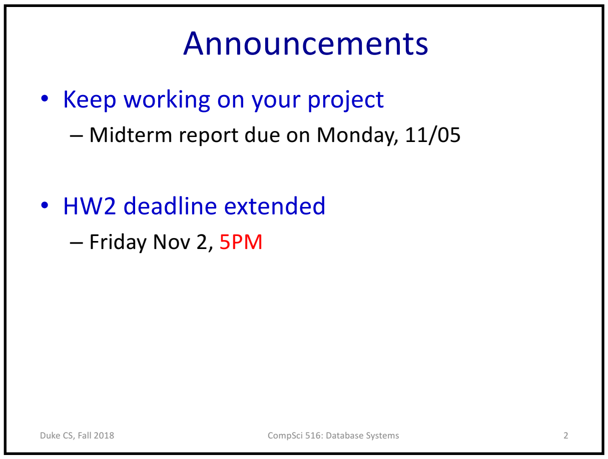
CompSci 516

Database Systems

Lecture 16

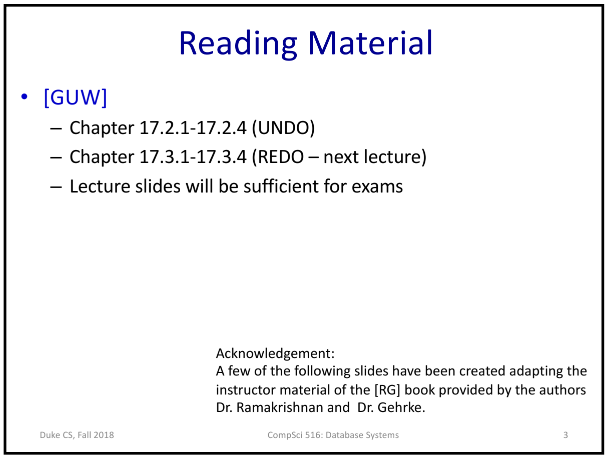
Transactions –
Recovery

Instructor: Sudeepa Roy



Announcements

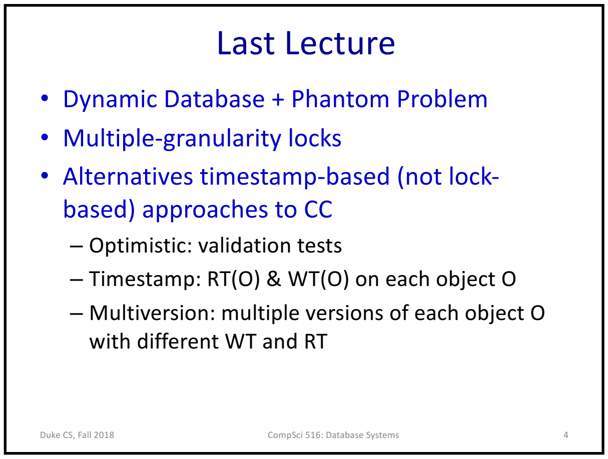
- Keep working on your project
  - Midterm report due on Monday, 11/05
- HW2 deadline extended
  - Friday Nov 2, 5PM



Reading Material

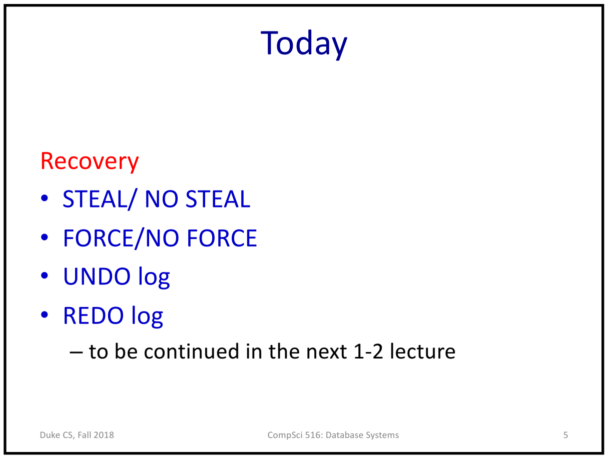
- [GUW]
  - Chapter 17.2.1-17.2.4 (UNDO)
  - Chapter 17.3.1-17.3.4 (REDO – next lecture)
  - Lecture slides will be sufficient for exams

Acknowledgement:
A few of the following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.



Last Lecture

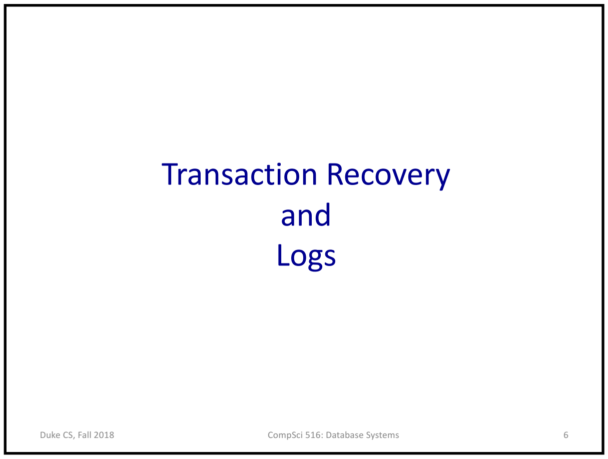
- Dynamic Database + Phantom Problem
- Multiple-granularity locks
- Alternatives timestamp-based (not lock-based) approaches to CC
  - Optimistic: validation tests
  - Timestamp: RT(O) & WT(O) on each object O
  - Multiversion: multiple versions of each object O with different WT and RT



Today

Recovery

- STEAL/ NO STEAL
- FORCE/NO FORCE
- UNDO log
- REDO log
  - to be continued in the next 1-2 lecture



Transaction Recovery and Logs

Review: The ACID properties

- **A** tomicity: All actions in the transaction happen, or none happen.
- **C** onsistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one transaction is isolated from that of other transactions.
- **D** urability: If a transaction commits, its effects persist.

- Which property did we cover in CC?

Review: The ACID properties

- **A** tomicity: All actions in the transaction happen, or none happen.
- **C** onsistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one transaction is isolated from that of other transactions.
- **D** urability: If a transaction commits, its effects persist.

- Which property did we cover in CC? : Isolation

Review: The ACID properties

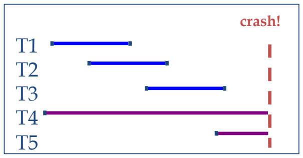
- **A** tomicity: All actions in the transaction happen, or none happen.
- **C** onsistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one transaction is isolated from that of other transactions.
- **D** urability: If a transaction commits, its effects persist.

- Next: The Recovery Manager guarantees Atomicity & Durability.
- Recall that Consistency is programmer's responsibility

Motivation: A & D

- Atomicity:
  - Transactions may abort ("Rollback").
- Durability:
  - What if DBMS stops running? (power failure/crash/error/fire-flood etc.)

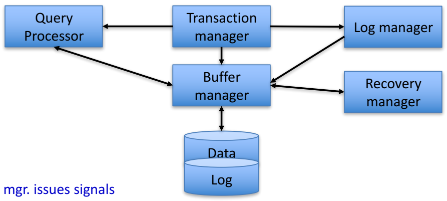
❖ Desired Behavior after system restarts:
- T1, T2 & T3 should be durable.
- T4 & T5 should be aborted (effects not seen).



Recovery: A & D

- Atomicity
  - by "undo"ing actions of "aborted transactions"
- Durability
  - by making sure that all actions of committed transactions survive crashes and system failure
  - i.e. by "redo"-ing actions of "committed transactions"

DB Architecture for Transactions



- Tr. mgr. issues signals
  - to log mgr. to store log records
  - to buffer mgr. about when it should copy buffer to disk
  - to query processor to execute queries/operations that comprise the transaction
- Log mgr.
  - maintains log, deals with buffer mgr. since first log appears in buffer then is written to disk
- Recovery mgr.
  - If there is a crash, it is activated
  - examines the log and repairs data if necessary
- Note: access to the disk is through the buffer mgr.

Assumptions

- Concurrency control is in effect
- Updates are happening "in place".
  - i.e. data is overwritten on (deleted from) the disk.
- Simple schemes to guarantee Atomicity & Durability (next):
  - NO STEAL
  - FORCE

Handling the Buffer Pool

- Force every write to disk?
  - Poor response time
  - But provides durability
- Steal buffer-pool frames from uncommitted transactions?
  - If not, poor throughput
  - If so, how can we ensure atomicity?

	No Steal	Steal
Force	Trivial	
No Force		Desired

What if we do "Steal" and "NO Force"

- STEAL (why enforcing Atomicity is hard)
  - To steal frame F: Current page in F (say P) is written to disk; some transaction holds lock on P
    - What if the transaction with the lock on P aborts?
    - Must remember the old value of P at steal time (to support UNDOing the write to page P)
- NO FORCE (why enforcing Durability is hard)
  - What if system crashes before a modified page is written to disk?
  - Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.

Basic Idea: Logging

- Record REDO and UNDO information, for every update, in a log
  - Sequential writes to log (put it on a separate disk)
  - Minimal info (diff) written to log, so multiple updates fit in a single log page
- Log: An ordered list of REDO/UNDO actions
  - Log record may contain:

    <Tr.ID, pageID, offset, length, old data, new data>

Different types of logs

- UNDO
- REDO
- UNDO/REDO

GUW 17.2, 17.3, 17.4
(Lecture material will be sufficient for HWs and Exams)

- ARIES
  - an UNDO/REDO log implementation

Next lecture, [RG]18,
and a research paper (optional)

Recall: Log Records

- A file opened for appending only
- Log blocks are created and updated in the main memory first
  - then written to disk

UNDO logging

UNDO logging

- Make repair to the database by undoing the effect of transactions that have not finished
  - i.e. uncommitted transactions before a crash or aborted transactions

Types of UNDO log records

- **<START T>:** transaction T has begun
- **<COMMIT T>:** T has completed successfully, no more changes will be made
  - Note that seeing <COMMIT T> does not automatically ensure that changes have been written to disk, has to be enforced by log manager
- **<ABORT T>:** transaction T could not complete successfully
  - job of the transaction mgr to ensure that changes by T never appear on disk or are cancelled
- **<T, X, v>:** update record for UNDO log
  - T has changed object X, and its former value was v
  - This change normally happened in memory after a WRITE, not after OUTPUT to disk
  - NOTE: we only record the old value, not the new value
  - since UNDO log, while undoing, replace with old value

UNDO logging rules

1. (U1) If T modifies X, then log record <T, X, v> must be written to disk before the new value of X is written to disk
   - so that the update can be undone using v if there is a crash

2. (U2) If T commits, <COMMIT T> must be written to disk after all database elements changed by T are written to disk
   - but as soon thereafter as possible

Order of write to disk for UNDO log

- Summarizing two rules:

for each element, not as a group

1. First, the log records indicating changed DB elements should be written

2. Second, the changed values of the DB elements should be written

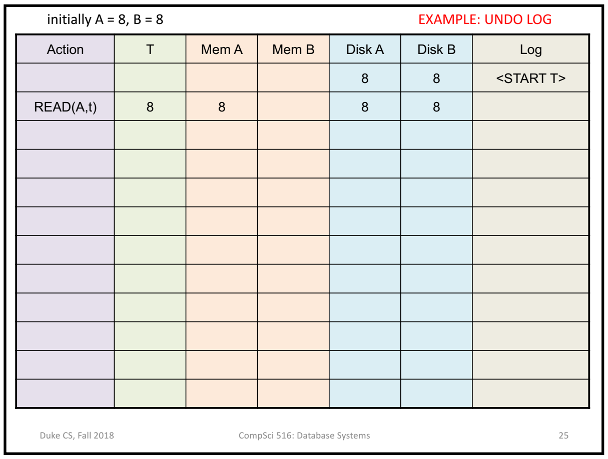
3. Finally, the COMMIT log record should be written

initially A = 8, B = 8

= t

EXAMPLE: UNDO LOG

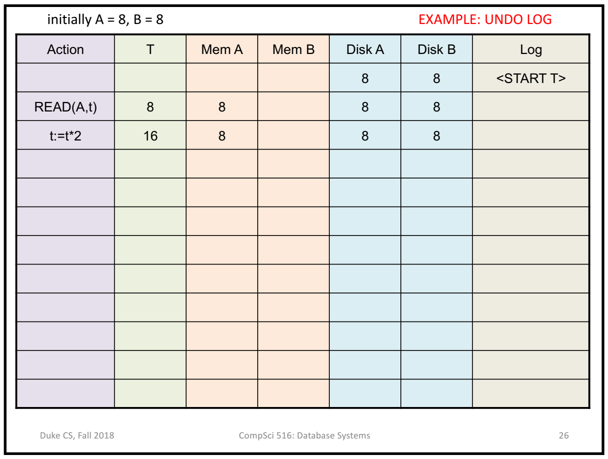
Action	T	Mem A	Mem B	Disk A	Disk B	Log
				8	8	<START T>



initially A = 8, B = 8

EXAMPLE: UNDO LOG

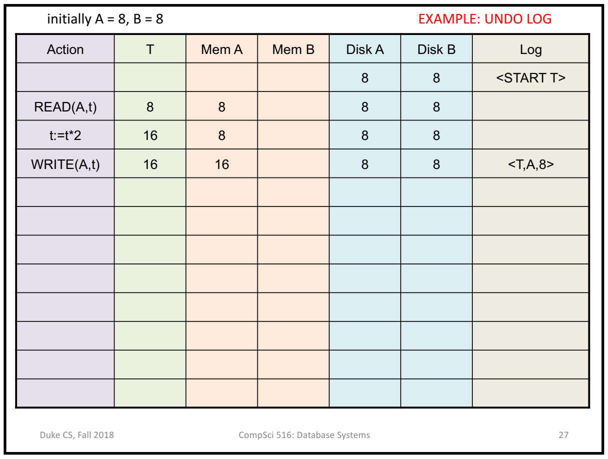
Action	T	Mem A	Mem B	Disk A	Disk B	Log
				8	8	<START T>
READ(A,t)	8	8		8	8	



initially A = 8, B = 8

EXAMPLE: UNDO LOG

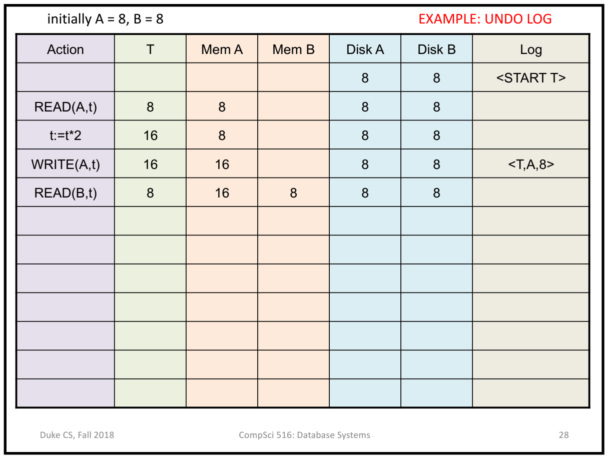
Action	T	Mem A	Mem B	Disk A	Disk B	Log
				8	8	<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	



initially A = 8, B = 8

EXAMPLE: UNDO LOG

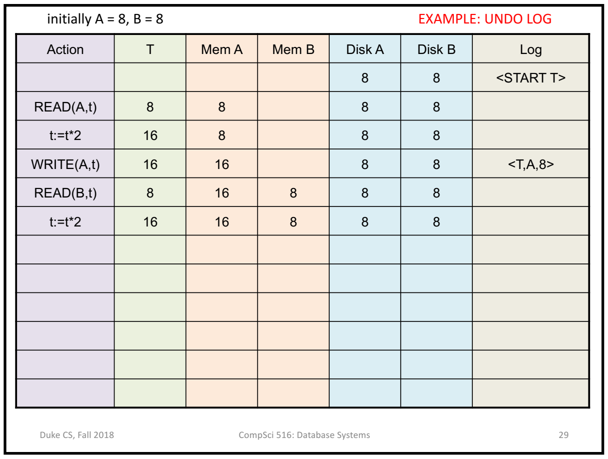
Action	T	Mem A	Mem B	Disk A	Disk B	Log
				8	8	<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>



initially A = 8, B = 8

EXAMPLE: UNDO LOG

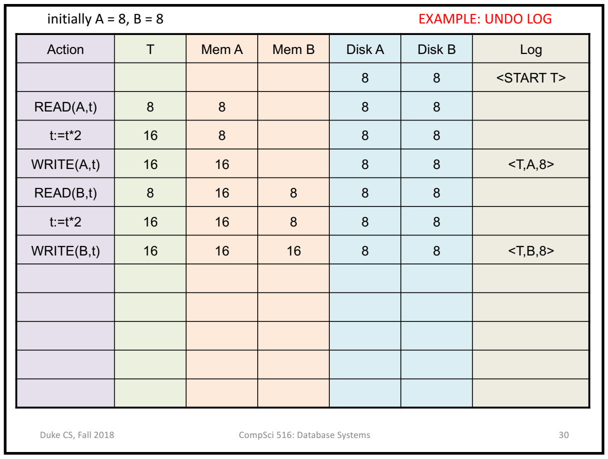
Action	T	Mem A	Mem B	Disk A	Disk B	Log
				8	8	<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	



initially A = 8, B = 8

EXAMPLE: UNDO LOG

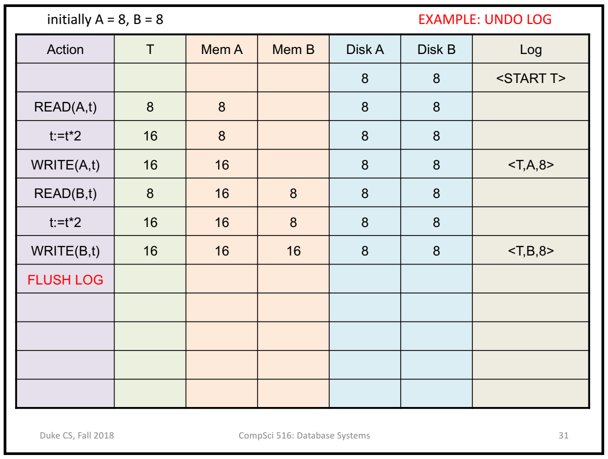
Action	T	Mem A	Mem B	Disk A	Disk B	Log
				8	8	<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	



initially A = 8, B = 8

EXAMPLE: UNDO LOG

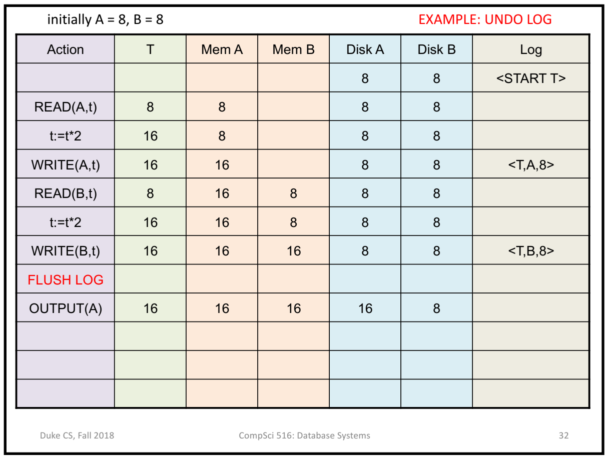
Action	T	Mem A	Mem B	Disk A	Disk B	Log
				8	8	<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>



initially A = 8, B = 8

EXAMPLE: UNDO LOG

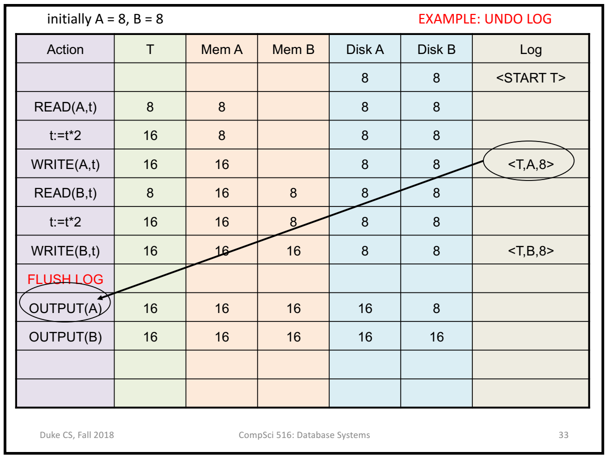
Action	T	Mem A	Mem B	Disk A	Disk B	Log
				8	8	<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH LOG						



initially A = 8, B = 8

EXAMPLE: UNDO LOG

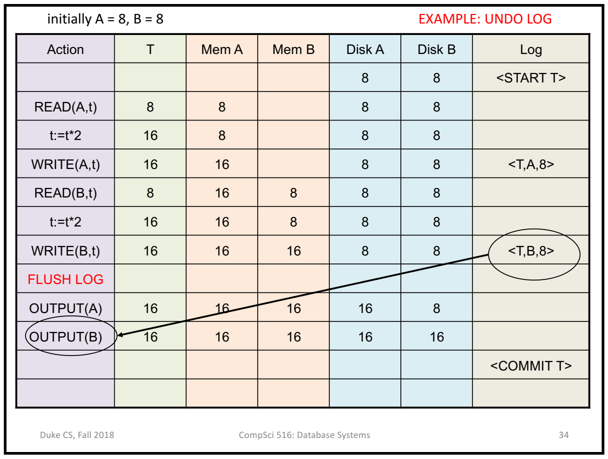
Action	T	Mem A	Mem B	Disk A	Disk B	Log
				8	8	<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	



initially A = 8, B = 8

EXAMPLE: UNDO LOG

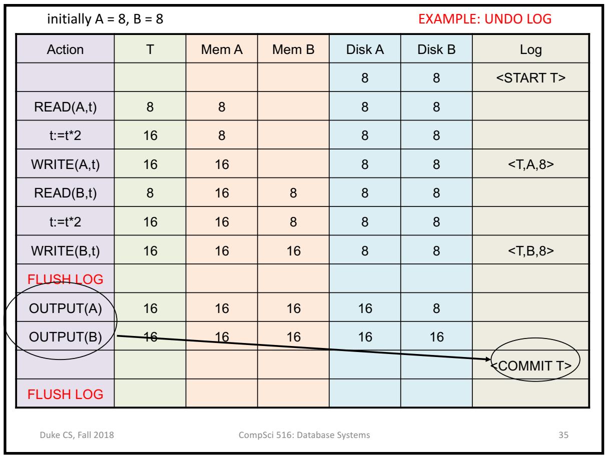
Action	T	Mem A	Mem B	Disk A	Disk B	Log
				8	8	<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	



initially A = 8, B = 8

EXAMPLE: UNDO LOG

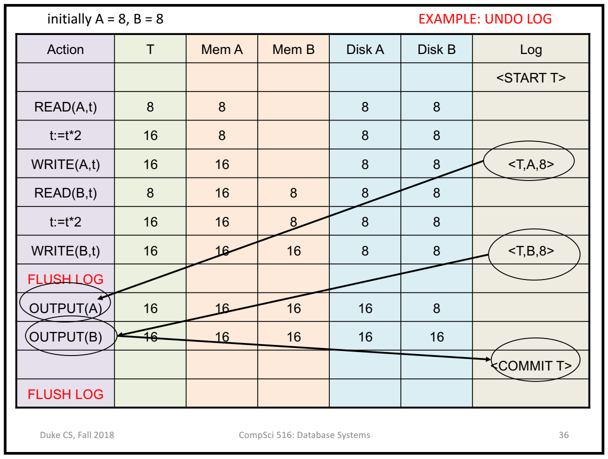
Action	T	Mem A	Mem B	Disk A	Disk B	Log
				8	8	<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>



initially A = 8, B = 8

EXAMPLE: UNDO LOG

Action	T	Mem A	Mem B	Disk A	Disk B	Log
				8	8	<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						



initially A = 8, B = 8

EXAMPLE: UNDO LOG

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Recovery using UNDO logging

- first simple (look at entire log)
- then "checkpointing" (no need to look at entire log)

Recovery with UNDO log

- Scan from the end
- If <COMMIT T> is found in log
  - all changes by T have been written to disk – OK
- <START T> found but no <COMMIT T>
  - some changes might be written, some not
  - Changes by T on disk have to be UNDONE
- Recall rule 1:
  - "If T modifies X, then log record <T, X, v> must be written to disk before the new value of X is written to disk"
  - v was previous value of X
  - For each such change on disk, there will be a log record on disk as well
  - Reset value of X to v in recovery

UNDO: order of writing to disk

1. <START T>
2. <T, A, 10> (old value 10)
3. A = 12 (new value 12)
4. <COMMIT T>

Recovery with UNDO log

- Travel backward
  - scan the log from the end toward the start
- Remember whether you have seen <COMMIT T> or <ABORT T> for all T
- Suppose <T, X, v> is encountered

1. If <COMMIT T> has been seen, do nothing
   - nothing to undo, new value already written
2. Otherwise,
   a) T is incomplete or aborted
   b) Change the value of X to v
3. If <ABORT T> not found
   a) write <ABORT T>
   b) flush the log
   c) resume normal operation

UNDO: order of writing to disk

1. <START T>
2. <T, A, 10> (old value 10)
3. A = 12 (new value 12)
4. <COMMIT T>

initially A = 8, B = 8 EXAMPLE: UNDO LOG Crash example 1

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

- Crash after final flush
- <COMMIT T> already on disks
- All log records by T are ignored by the recovery manager

initially A = 8, B = 8 EXAMPLE: UNDO LOG Crash example 2, Step 1

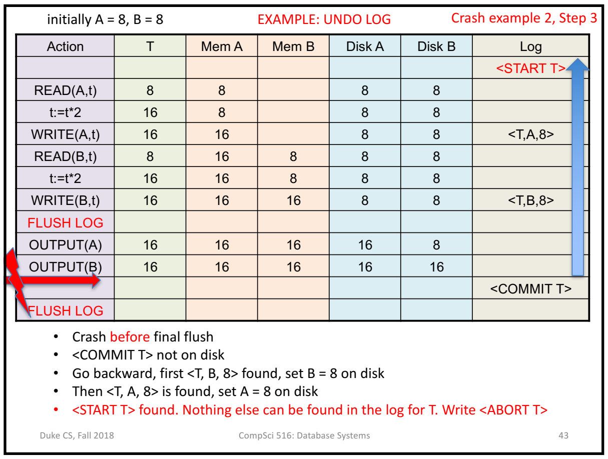
Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

- Crash before final flush
- <COMMIT T> not on disk
- Go backward, first <T, B, 8> found, set B = 8 on disk

initially A = 8, B = 8 EXAMPLE: UNDO LOG Crash example 2, Step 2

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

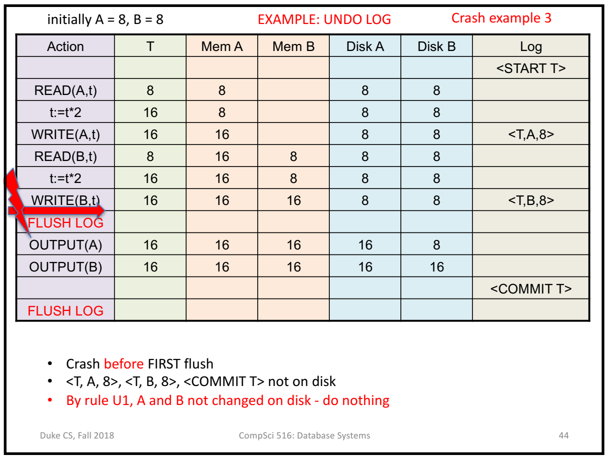
- Crash before final flush
- <COMMIT T> not on disk
- Go backward, first <T, B, 8> found, set B = 8 on disk
- Then <T, A, 8> is found, set A = 8 on disk



initially A = 8, B = 8 EXAMPLE: UNDO LOG Crash example 2, Step 3

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

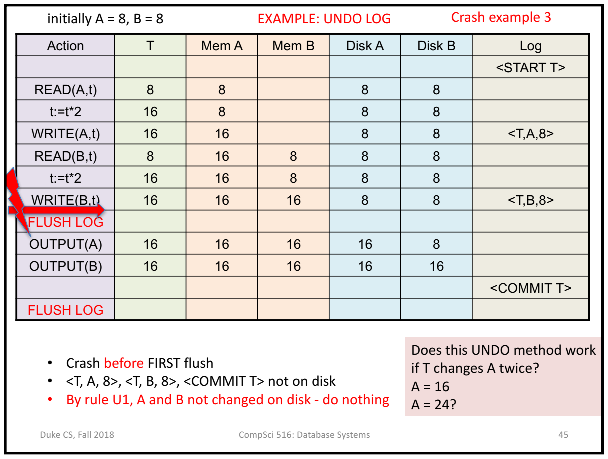
- Crash before final flush
- <COMMIT T> not on disk
- Go backward, first <T, B, 8> found, set B = 8 on disk
- Then <T, A, 8> is found, set A = 8 on disk
- <START T> found. Nothing else can be found in the log for T. Write <ABORT T>



initially A = 8, B = 8 EXAMPLE: UNDO LOG Crash example 3

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

- Crash before FIRST flush
- <T, A, 8>, <T, B, 8>, <COMMIT T> not on disk
- By rule U1, A and B not changed on disk - do nothing

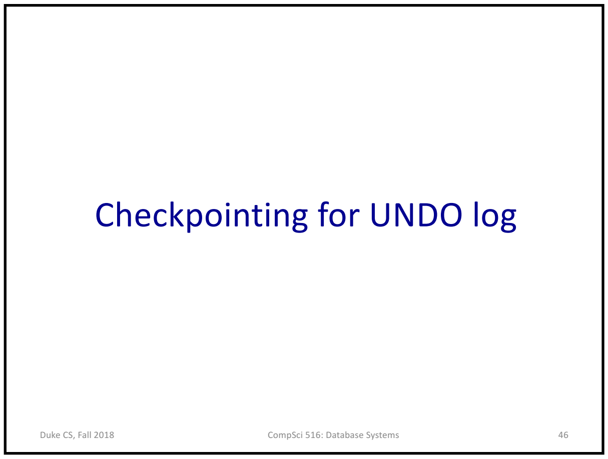


initially A = 8, B = 8 EXAMPLE: UNDO LOG Crash example 3

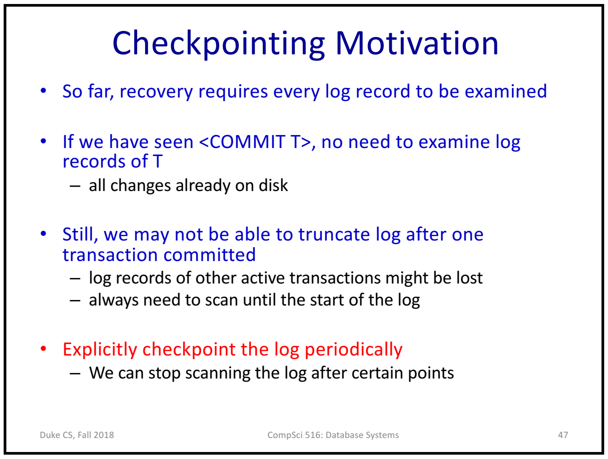
Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

- Crash before FIRST flush
- <T, A, 8>, <T, B, 8>, <COMMIT T> not on disk
- By rule U1, A and B not changed on disk - do nothing

Does this UNDO method work if T changes A twice?
A = 16
A = 24?

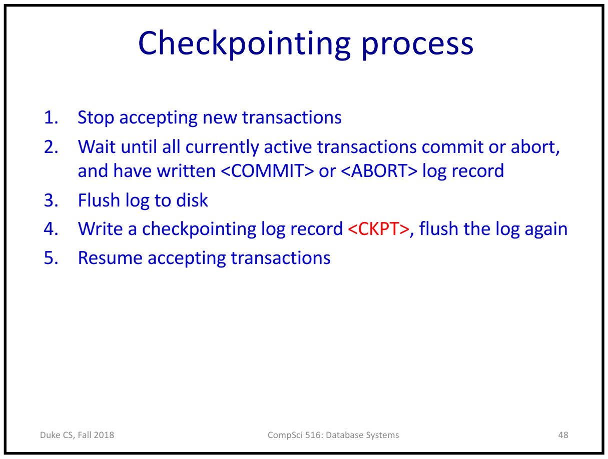


Checkpointing for UNDO log



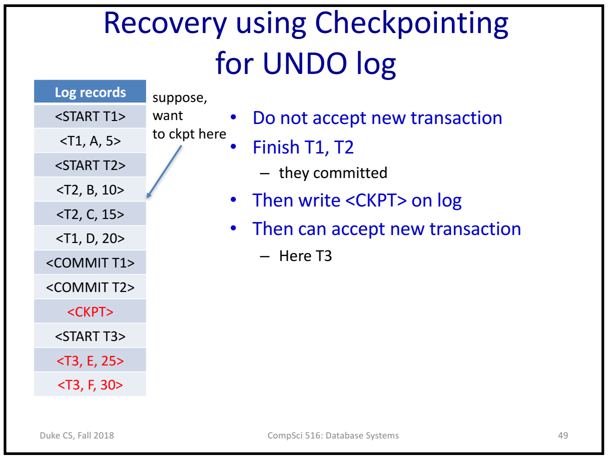
Checkpointing Motivation

- So far, recovery requires every log record to be examined
- If we have seen <COMMIT T>, no need to examine log records of T
  - all changes already on disk
- Still, we may not be able to truncate log after one transaction committed
  - log records of other active transactions might be lost
  - always need to scan until the start of the log
- Explicitly checkpoint the log periodically
  - We can stop scanning the log after certain points



Checkpointing process

1. Stop accepting new transactions
2. Wait until all currently active transactions commit or abort, and have written <COMMIT> or <ABORT> log record
3. Flush log to disk
4. Write a checkpointing log record <CKPT>, flush the log again
5. Resume accepting transactions

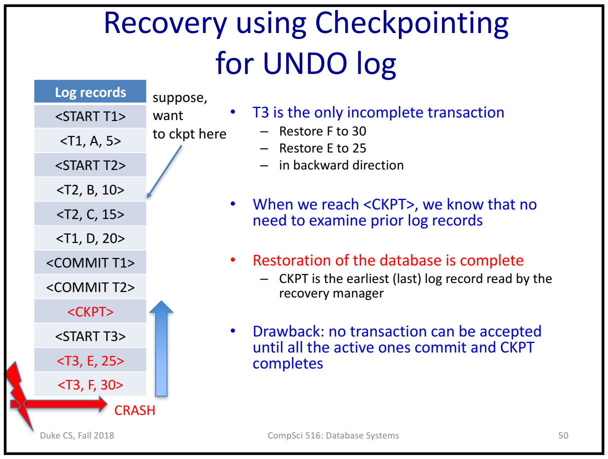


Recovery using Checkpointing for UNDO log

Log records
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<T2, C, 15>
<T1, D, 20>
<COMMIT T1>
<COMMIT T2>
<CKPT>
<START T3>
<T3, E, 25>
<T3, F, 30>

suppose, want to ckpt here

- Do not accept new transaction
- Finish T1, T2
  - they committed
- Then write <CKPT> on log
- Then can accept new transaction
  - Here T3



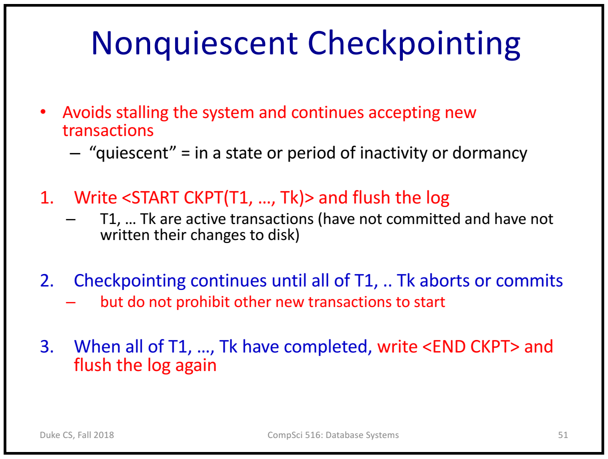
Recovery using Checkpointing for UNDO log

Log records
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<T2, C, 15>
<T1, D, 20>
<COMMIT T1>
<COMMIT T2>
<CKPT>
<START T3>
<T3, E, 25>
<T3, F, 30>

suppose, want to ckpt here

CRASH

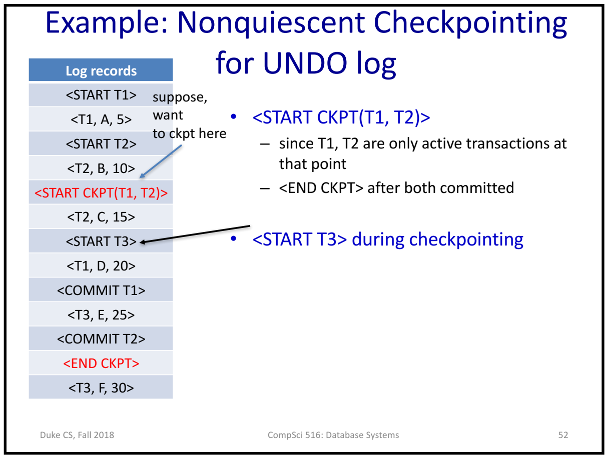
- T3 is the only incomplete transaction
  - Restore F to 30
  - Restore E to 25
  - in backward direction
- When we reach <CKPT>, we know that no need to examine prior log records
- Restoration of the database is complete
  - CKPT is the earliest (last) log record read by the recovery manager
- Drawback: no transaction can be accepted until all the active ones commit and CKPT completes



Nonquiescent Checkpointing

- Avoids stalling the system and continues accepting new transactions
  - "quiescent" = in a state or period of inactivity or dormancy

1. Write <START CKPT(T1, …, Tk)> and flush the log
   - T1, … Tk are active transactions (have not committed and have not written their changes to disk)
2. Checkpointing continues until all of T1, .. Tk aborts or commits
   - but do not prohibit other new transactions to start
3. When all of T1, …, Tk have completed, write <END CKPT> and flush the log again

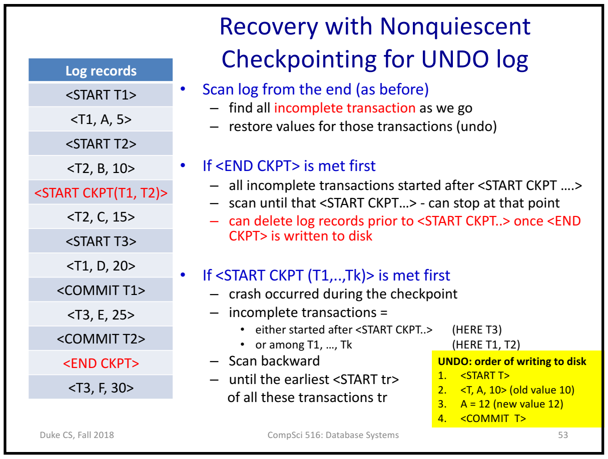


Example: Nonquiescent Checkpointing for UNDO log

Log records
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT(T1, T2)>
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

suppose, want to ckpt here

- <START CKPT(T1, T2)>
  - since T1, T2 are only active transactions at that point
  - <END CKPT> after both committed
- <START T3> during checkpointing

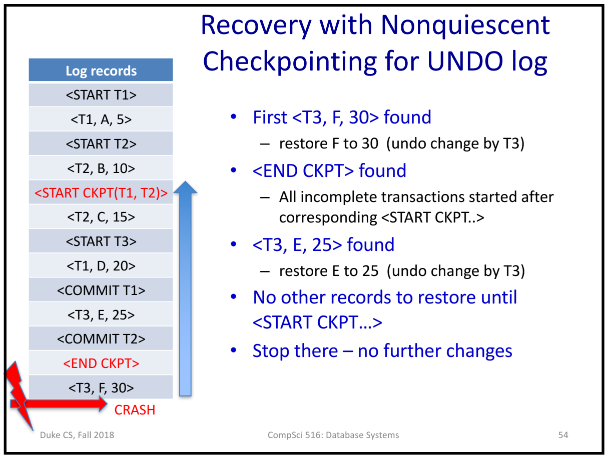


Recovery with Nonquiescent Checkpointing for UNDO log

Log records
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT(T1, T2)>
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

- Scan log from the end (as before)
  - find all incomplete transaction as we go
  - restore values for those transactions (undo)
- If <END CKPT> is met first
  - all incomplete transactions started after <START CKPT ….>
  - scan until that <START CKPT…> - can stop at that point
  - can delete log records prior to <START CKPT..> once <END CKPT> is written to disk
- If <START CKPT (T1,..,Tk)> is met first
  - crash occurred during the checkpoint
  - incomplete transactions =
    - either started after <START CKPT..> (HERE T3)
    - or among T1, …, Tk (HERE T1, T2)
  - Scan backward
  - until the earliest <START tr> of all these transactions tr

UNDO: order of writing to disk
1. <START T>
2. <T, A, 10> (old value 10)
3. A = 12 (new value 12)
4. <COMMIT T>



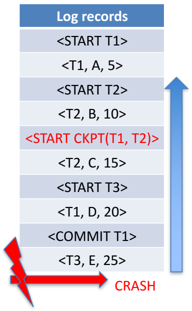
Recovery with Nonquiescent Checkpointing for UNDO log

Log records
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT(T1, T2)>
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

CRASH

- First <T3, F, 30> found
  - restore F to 30 (undo change by T3)
- <END CKPT> found
  - All incomplete transactions started after corresponding <START CKPT..>
- <T3, E, 25> found
  - restore E to 25 (undo change by T3)
- No other records to restore until <START CKPT…>
- Stop there – no further changes

Recovery with Nonquiescent Checkpointing for UNDO log



Log records
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT(T1, T2)>
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>
<T3, E, 25>

CRASH

- Scan backward
  - no <END CKPT> found
  - but <START CKPT(T1, T2)> found
  - also <COMMIT T1> found
- T3 and T2 incomplete transactions
  - T1 already committed
- Scan until the earliest of <START T2> and <START T3>
  - here <START T2>
- Along the way backward
  - restore E to 25 (undo change by T3)
  - restore C to 15 (undo change by T2)
  - restore B to 10 (undo change by T2)
  - in this order
  - then stop at <START T2>

Problems with UNDO logging

- We cannot commit T unless all its changes appear on disk
- Sometimes disk I/Os can be saved if the changes can stay in main memory for a while
  - as long as there is a log to fix things in a crash
- Idea: REDO logging

UNDO: order of writing to disk
1. <START T>
2. <T, A, 10> (old value 10)
3. A = 12 (new value 12)
4. <COMMIT T>

REDO logging

Review: UNDO Log

Considered by UNDO log
UNDO uncommitted transactions
Ignore committed transactions

- STEAL
  - to be able to steal modified pages by a running transaction
  - may have to UNDO for uncommitted transactions

Considered by REDO log
REDO committed transactions
Ignore uncommitted transactions

- NO FORCE
  - not to force every write of running transaction to disk
  - may have to REDO for committed transactions

UNDO vs. REDO

UNDO	REDO
cancels (UNDO) the effect of incomplete transactions	ignores incomplete transactions
ignores committed ones	repeats (REDO) the changes made by committed ones
requires writing changed elements to disk BEFORE the commit log record is written	requires writing changed elements to disk AFTER the commit log record is written
<T, X, v>: v is OLD value of X	<T, X v>: v is NEW value of X

Types of REDO log records

same as UNDO

- <START T>
  - transaction T has begun
- <COMMIT T>
  - T has completed successfully, no more changes will be made
  - Note that seeing <COMMIT T> does not automatically ensure that changes have been written to disk
    - has to be enforced by log manager
- <ABORT T>
  - transaction T could not complete successfully
  - job of the transaction mgr to ensure that changes by T never appear on disk or are cancelled
- <T, X, v>
  - update record for REDO log
  - T has changed object X, and its NEW value is v
- NOTE: we only record the new value, not the old value (unlike UNDO)

REDO logging rule

(R1) Before modifying any element X on disk, all log records pertaining to this modification, including <T, X, v> and <COMMIT T>, must appear on disk

- single “redo rule”
- called the WRITE-AHEAD LOGGING (WAL) rule

Order of write to disk for REDO log

1. First, the log records indicating changed DB elements should be written
2. Second, The COMMIT log record should be written
3. Finally, the changed DB elements should be written

UNDO: order of writing to disk

1. <START T>
2. <T, A, 10> (old value 10)
3. A = 12 (new value 12)
4. <COMMIT T>

REDO: order of writing to disk

1. <START T>
2. <T, A, 12> (new value 12)
3. <COMMIT T>
4. A = 12 (new value 12)

different order than UNDO

initially A = 8, B = 8

EXAMPLE: REDO LOG

	Action	T	Mem A	Mem B	Disk A	Disk B	Log
1							<START T>
2	READ(A,t)	8	8		8	8	
3	t:=t*2	16	8		8	8	
4	WRITE(A,t)	16	16		8	8	<T, A, 16>
5	READ(B,t)	8	16	8	8	8	
6	t:=t*2	16	16	8	8	8	
7	WRITE(B,t)	16	16	16	8	8	<T,B,16>
8							<COMMIT T>
9	FLUSH LOG						
10	OUTPUT(A)	16	16	16	16	8	
11	OUTPUT(B)	16	16	16	16	16	

initially A = 8, B = 8

EXAMPLE: REDO LOG

	Action	T	Mem A	Mem B	Disk A	Disk B	Log
1							<START T>
2	READ(A,t)	8	8		8	8	
3	t:=t*2	16	8		8	8	
4	WRITE(A,t)	16	16		8	8	<T, A, 16>
5	READ(B,t)	8	16	8	8	8	
6	t:=t*2	16	16	8	8	8	
7	WRITE(B,t)	16	16	16	8	8	<T, B,16>
8							<COMMIT T>
9	FLUSH LOG						
10	OUTPUT(A)	16	16	16	16	8	
11	OUTPUT(B)	16	16	16	16	16	

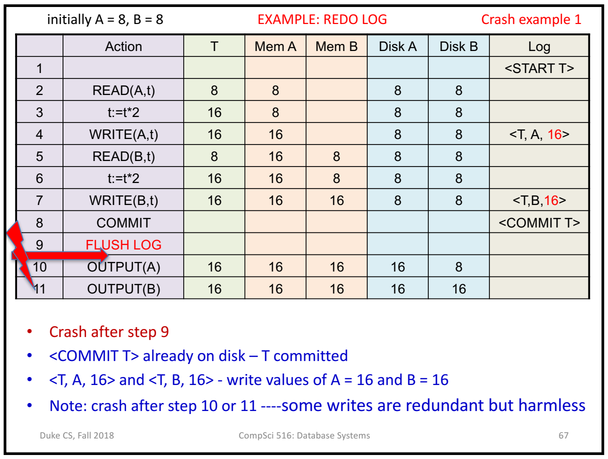
Recovery using REDO logging

Recovery with REDO log

- Identify committed transactions
  - scan from the end to identify committed transactions
  - make redo changes in the forward direction
- For each log record <T, X, v>
  - If T is not a committed transaction
    - do nothing
  - If T is committed
    - write the value v of element X
- For each incomplete transaction T
  - write <ABORT T>
  - Flush the log

REDO: order of writing to disk

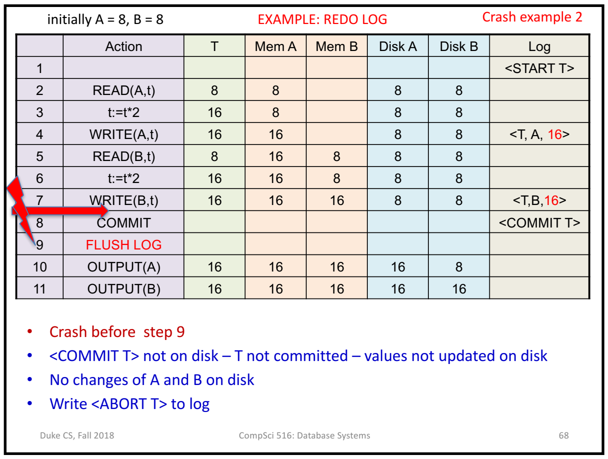
1. <START T>
2. <T, A, 12> (new value 12)
3. <COMMIT T>
4. A = 12 (new value 12)



initially A = 8, B = 8 EXAMPLE: REDO LOG Crash example 1

	Action	T	Mem A	Mem B	Disk A	Disk B	Log
1							<START T>
2	READ(A,t)	8	8		8	8	
3	t:=t*2	16	8		8	8	
4	WRITE(A,t)	16	16		8	8	<T, A, 16>
5	READ(B,t)	8	16	8	8	8	
6	t:=t*2	16	16	8	8	8	
7	WRITE(B,t)	16	16	16	8	8	<T,B,16>
8	COMMIT						<COMMIT T>
9	FLUSH LOG						
10	OUTPUT(A)	16	16	16	16	8	
11	OUTPUT(B)	16	16	16	16	16	

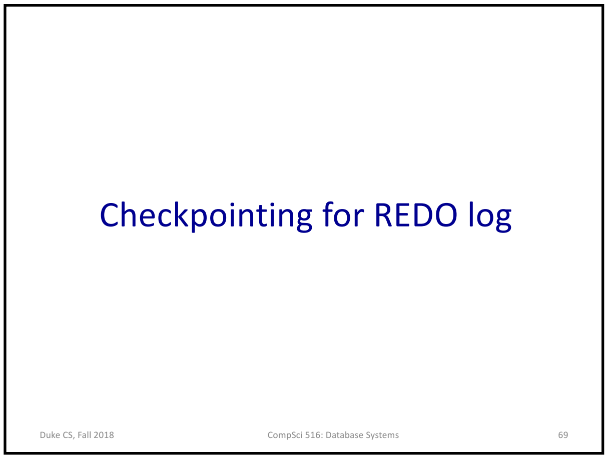
- Crash after step 9
- <COMMIT T> already on disk – T committed
- <T, A, 16> and <T, B, 16> - write values of A = 16 and B = 16
- Note: crash after step 10 or 11 ----some writes are redundant but harmless



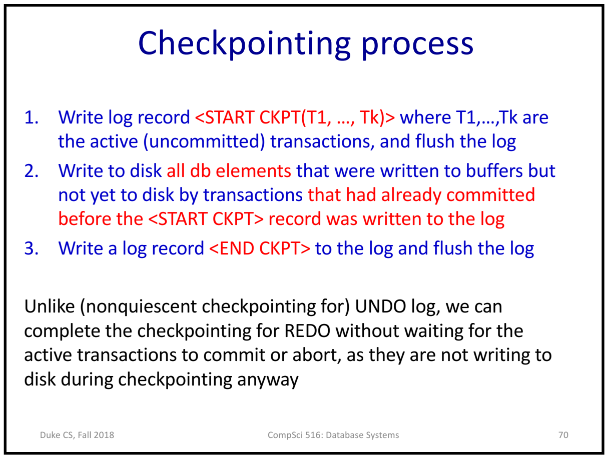
initially A = 8, B = 8 EXAMPLE: REDO LOG Crash example 2

	Action	T	Mem A	Mem B	Disk A	Disk B	Log
1							<START T>
2	READ(A,t)	8	8		8	8	
3	t:=t*2	16	8		8	8	
4	WRITE(A,t)	16	16		8	8	<T, A, 16>
5	READ(B,t)	8	16	8	8	8	
6	t:=t*2	16	16	8	8	8	
7	WRITE(B,t)	16	16	16	8	8	<T,B,16>
8	COMMIT						<COMMIT T>
9	FLUSH LOG						
10	OUTPUT(A)	16	16	16	16	8	
11	OUTPUT(B)	16	16	16	16	16	

- Crash before step 9
- <COMMIT T> not on disk – T not committed – values not updated on disk
- No changes of A and B on disk
- Write <ABORT T> to log



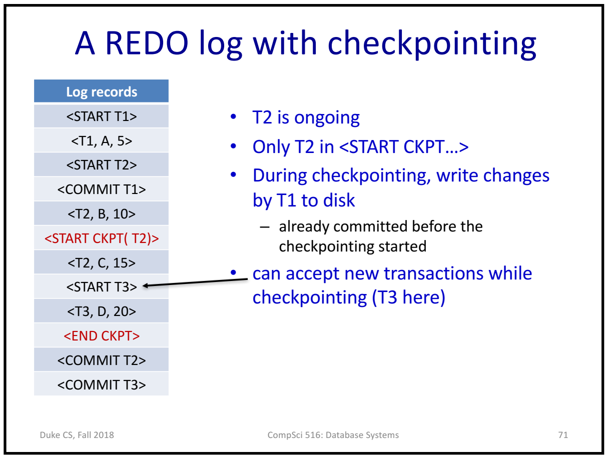
Checkpointing for REDO log



Checkpointing process

1. Write log record <START CKPT(T1, …, Tk)> where T1,…,Tk are the active (uncommitted) transactions, and flush the log
2. Write to disk all db elements that were written to buffers but not yet to disk by transactions that had already committed before the <START CKPT> record was written to the log
3. Write a log record <END CKPT> to the log and flush the log

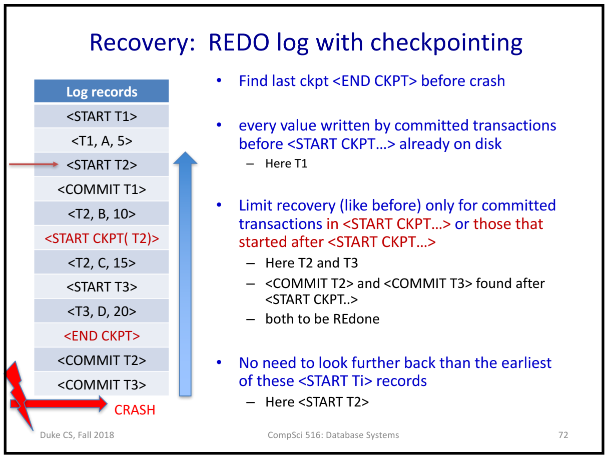
Unlike (nonquiescent checkpointing for) UNDO log, we can complete the checkpointing for REDO without waiting for the active transactions to commit or abort, as they are not writing to disk during checkpointing anyway



A REDO log with checkpointing

Log records
<START T1>
<T1, A, 5>
<START T2>
<COMMIT T1>
<T2, B, 10>
<START CKPT(T2)>
<T2, C, 15>
<START T3>
<T3, D, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

- T2 is ongoing
- Only T2 in <START CKPT…>
- During checkpointing, write changes by T1 to disk
  - already committed before the checkpointing started
- can accept new transactions while checkpointing (T3 here)

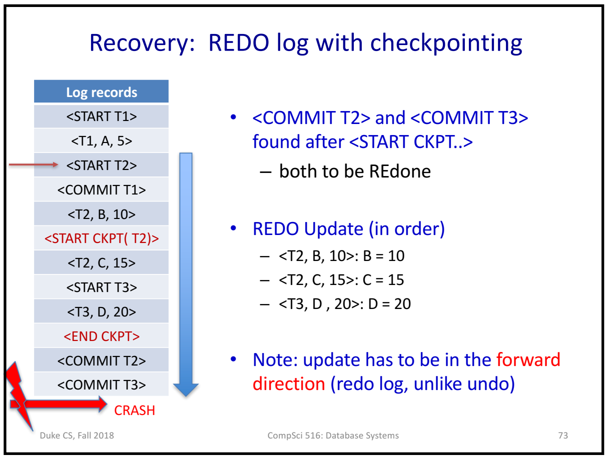


Recovery: REDO log with checkpointing

Log records
<START T1>
<T1, A, 5>
<START T2>
<COMMIT T1>
<T2, B, 10>
<START CKPT(T2)>
<T2, C, 15>
<START T3>
<T3, D, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

CRASH

- Find last ckpt <END CKPT> before crash
- every value written by committed transactions before <START CKPT…> already on disk
  - Here T1
- Limit recovery (like before) only for committed transactions in <START CKPT…> or those that started after <START CKPT…>
  - Here T2 and T3
  - <COMMIT T2> and <COMMIT T3> found after <START CKPT..>
  - both to be REdone
- No need to look further back than the earliest of these <START Ti> records
  - Here <START T2>

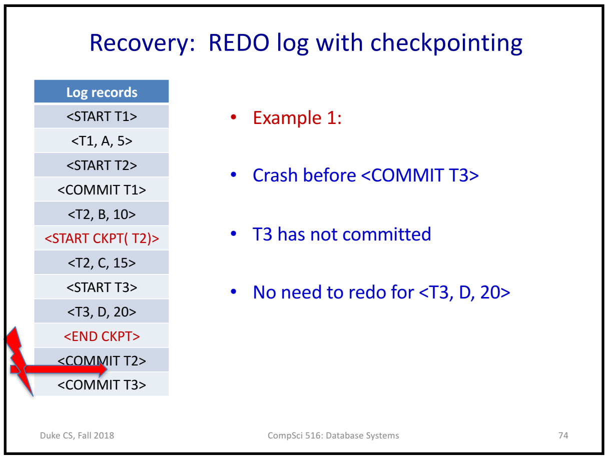


Recovery: REDO log with checkpointing

Log records
<START T1>
<T1, A, 5>
<START T2>
<COMMIT T1>
<T2, B, 10>
<START CKPT(T2)>
<T2, C, 15>
<START T3>
<T3, D, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

CRASH

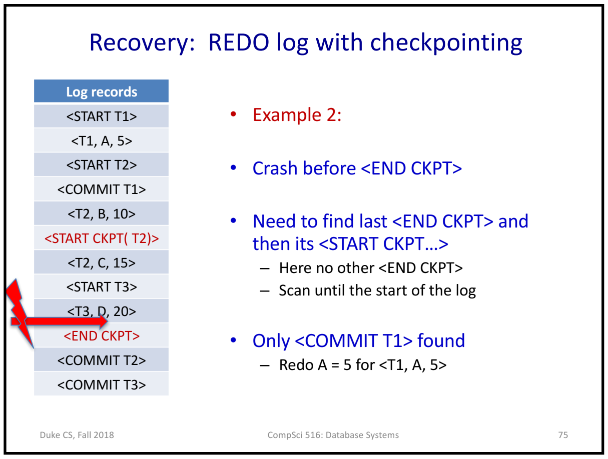
- <COMMIT T2> and <COMMIT T3> found after <START CKPT..>
  - both to be REdone
- REDO Update (in order)
  - <T2, B, 10>: B = 10
  - <T2, C, 15>: C = 15
  - <T3, D , 20>: D = 20
- Note: update has to be in the forward direction (redo log, unlike undo)



Recovery: REDO log with checkpointing

Log records
<START T1>
<T1, A, 5>
<START T2>
<COMMIT T1>
<T2, B, 10>
<START CKPT(T2)>
<T2, C, 15>
<START T3>
<T3, D, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

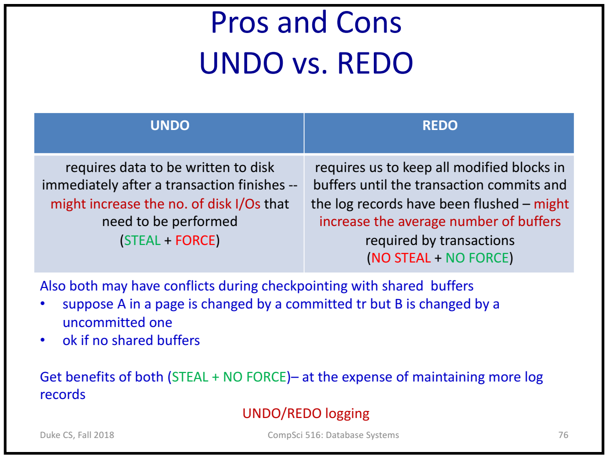
- Example 1:
- Crash before <COMMIT T3>
- T3 has not committed
- No need to redo for <T3, D, 20>



Recovery: REDO log with checkpointing

Log records
<START T1>
<T1, A, 5>
<START T2>
<COMMIT T1>
<T2, B, 10>
<START CKPT(T2)>
<T2, C, 15>
<START T3>
<T3, D, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

- Example 2:
- Crash before <END CKPT>
- Need to find last <END CKPT> and then its <START CKPT…>
  - Here no other <END CKPT>
  - Scan until the start of the log
- Only <COMMIT T1> found
  - Redo A = 5 for <T1, A, 5>



Pros and Cons
UNDO vs. REDO

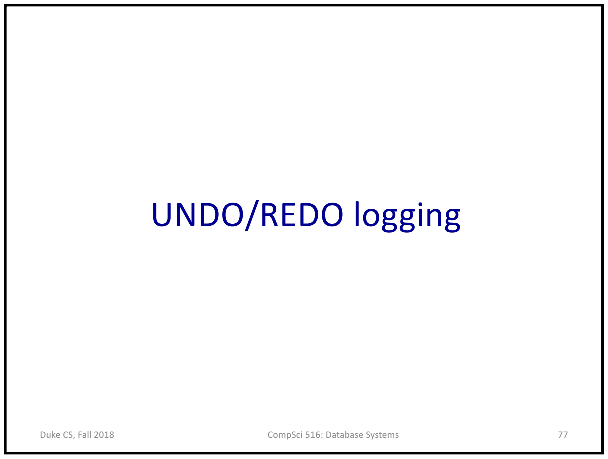
UNDO	REDO
requires data to be written to disk immediately after a transaction finishes -- might increase the no. of disk I/Os that need to be performed (STEAL + FORCE)	requires us to keep all modified blocks in buffers until the transaction commits and the log records have been flushed – might increase the average number of buffers required by transactions (NO STEAL + NO FORCE)

Also both may have conflicts during checkpointing with shared buffers

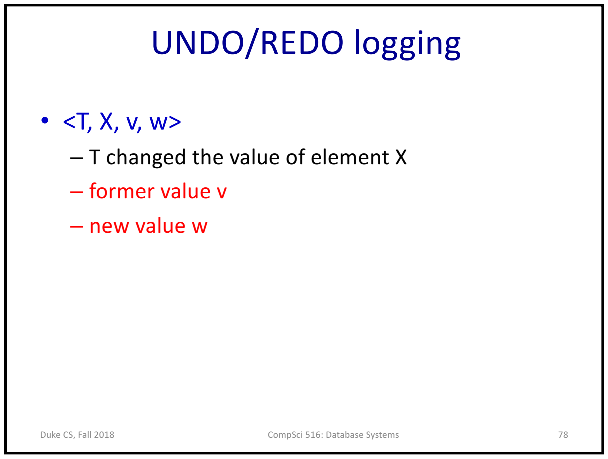
- suppose A in a page is changed by a committed tr but B is changed by a uncommitted one
- ok if no shared buffers

Get benefits of both (STEAL + NO FORCE)– at the expense of maintaining more log records

UNDO/REDO logging



UNDO/REDO logging



UNDO/REDO logging

- <T, X, v, w>
  - T changed the value of element X
  - former value v
  - new value w

UNDO/REDO logging rule

(UR rule) Before modifying any element X on disk, <T, X, v, w> must appear on disk

- Only constraint imposed by both UNDO and REDO log
- no constraint on <COMMIT T>
    - can precede or follow any of the changes to the db elements on disk

UNDO: order of writing to disk

1. <START T>
2. <T, A, 10> (old value 10)
3. A = 12 (new value 12)
4. <COMMIT T>

REDO: order of writing to disk

1. <START T>
2. <T, A, 12> (new value 12)
3. <COMMIT T>
4. A = 12 (new value 12)

initially A = 8, B = 8

EXAMPLE: UNDO/REDO LOG

	Action	T	Mem A	Mem B	Disk A	Disk B	Log
1							<START T>
2	READ(A,t)	8	8		8	8	
3	t:=t*2	16	8		8	8	
4	WRITE(A,t)	16	16		8	8	<T, A, 8,16>
5	READ(B,t)	8	16	8	8	8	
6	t:=t*2	16	16	8	8	8	
7	WRITE(B,t)	16	16	16	8	8	<T, B, 8,16>
8	FLUSH LOG						
9	OUTPUT(A)	16	16	16	16	8	
10							<COMMIT T>
11	OUTPUT(B)	16	16	16	16	16	

Step 10 (commit) could have also appeared before Step (8), before Step (9), or after Step (11)

Recovery using UNDO/REDO logging

Recovery with UNDO/REDO log

- REDO all committed transactions in the order earliest-first (forward)
- UNDO all uncommitted/incomplete transactions in the order latest first (backward)

- Important to do both
    - because of the flexibility allowed by UNDO/REDO logging regarding <COMMIT> records
- we can have
    - a committed transaction with not all changes written to disk
    - an uncommitted transactions with some changes written to disk

initially A = 8, B = 8

EXAMPLE: UNDO/REDO LOG

Crash example 1

	Action	T	Mem A	Mem B	Disk A	Disk B	Log
1							<START T>
2	READ(A,t)	8	8		8	8	
3	t:=t*2	16	8		8	8	
4	WRITE(A,t)	16	16		8	8	<T, A, 8,16>
5	READ(B,t)	8	16	8	8	8	
6	t:=t*2	16	16	8	8	8	
7	WRITE(B,t)	16	16	16	8	8	<T, B, 8,16>
8	FLUSH LOG						
9	OUTPUT(A)	16	16	16	16	8	
10							<COMMIT T>
11	OUTPUT(B)	16	16	16	16	16	

- Crash after <COMMIT T > is flushed to disk
- T is considered as committed
- First update A to 16
- Then update B to 16 (forward direction)
- Some changes may be unnecessary but harmless

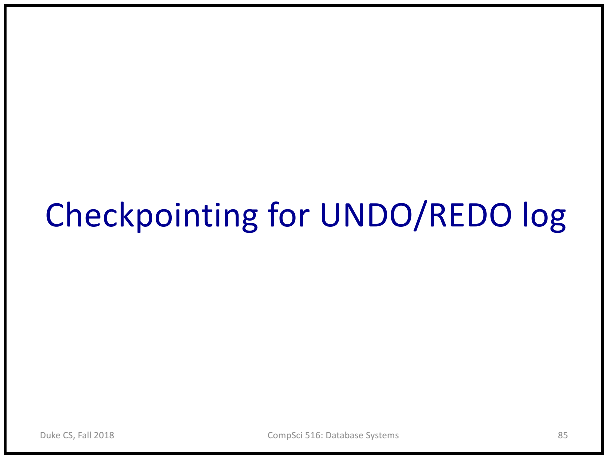
initially A = 8, B = 8

EXAMPLE: UNDO/REDO LOG

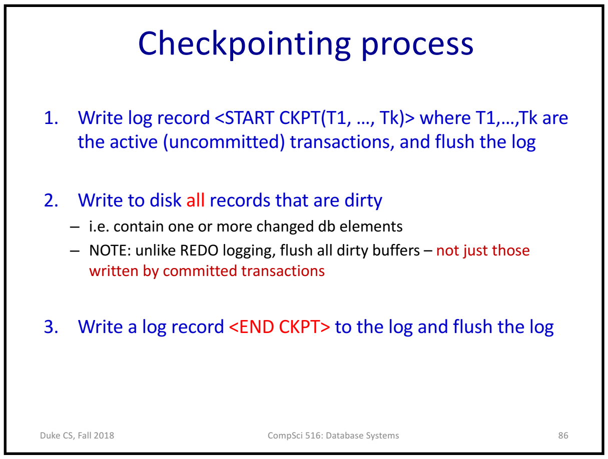
Crash example 1

	Action	T	Mem A	Mem B	Disk A	Disk B	Log
1							<START T>
2	READ(A,t)	8	8		8	8	
3	t:=t*2	16	8		8	8	
4	WRITE(A,t)	16	16		8	8	<T, A, 8,16>
5	READ(B,t)	8	16	8	8	8	
6	t:=t*2	16	16	8	8	8	
7	WRITE(B,t)	16	16	16	8	8	<T, B, 8,16>
8	FLUSH LOG						
9	OUTPUT(A)	16	16	16	16	8	
10							<COMMIT T>
11	OUTPUT(B)	16	16	16	16	16	

- Crash before <COMMIT T > is flushed to disk
- T is considered as uncommitted
- First update B to 8
- Then update A to 8 (backward direction)
- Some changes may be unnecessary but harmless

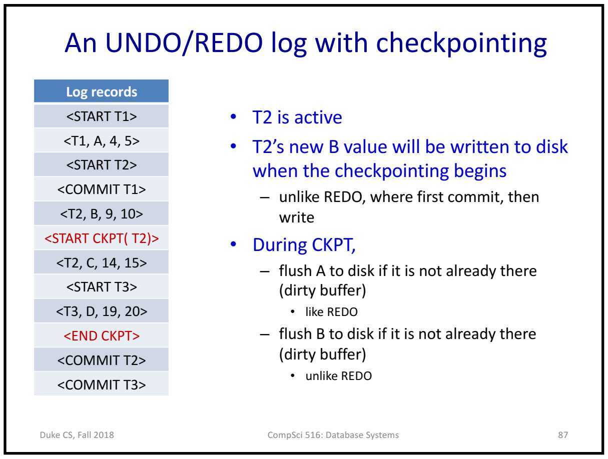


Checkpointing for UNDO/REDO log



Checkpointing process

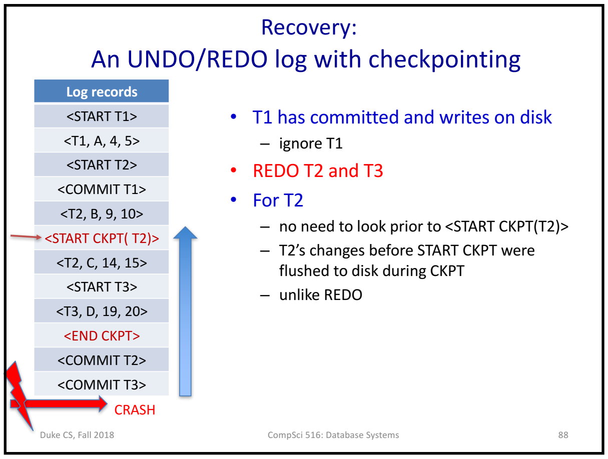
1. Write log record <START CKPT(T1, …, Tk)> where T1,…,Tk are the active (uncommitted) transactions, and flush the log
2. Write to disk all records that are dirty
   - i.e. contain one or more changed db elements
   - NOTE: unlike REDO logging, flush all dirty buffers – not just those written by committed transactions
3. Write a log record <END CKPT> to the log and flush the log



An UNDO/REDO log with checkpointing

Log records
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>
<T2, B, 9, 10>
<START CKPT(T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

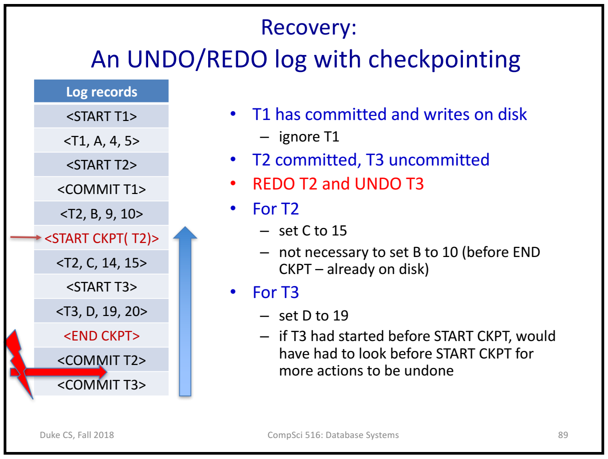
- T2 is active
- T2's new B value will be written to disk when the checkpointing begins
  - unlike REDO, where first commit, then write
- During CKPT,
  - flush A to disk if it is not already there (dirty buffer)
    - like REDO
  - flush B to disk if it is not already there (dirty buffer)
    - unlike REDO



Recovery:
An UNDO/REDO log with checkpointing

Log records
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>
<T2, B, 9, 10>
<START CKPT(T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

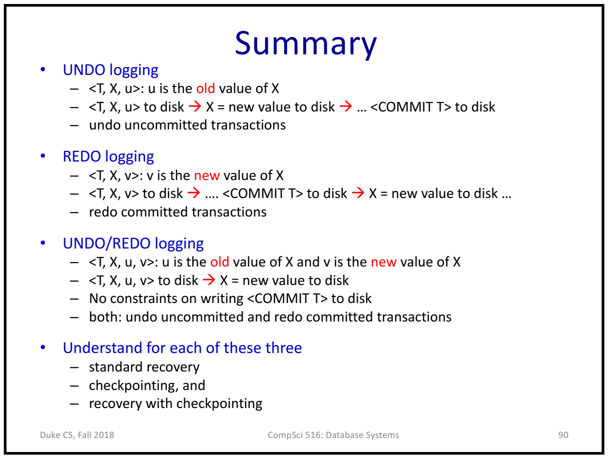
- T1 has committed and writes on disk
  - ignore T1
- REDO T2 and T3
- For T2
  - no need to look prior to <START CKPT(T2)>
  - T2's changes before START CKPT were flushed to disk during CKPT
  - unlike REDO



Recovery:
An UNDO/REDO log with checkpointing

Log records
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>
<T2, B, 9, 10>
<START CKPT(T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

- T1 has committed and writes on disk
  - ignore T1
- T2 committed, T3 uncommitted
- REDO T2 and UNDO T3
- For T2
  - set C to 15
  - not necessary to set B to 10 (before END CKPT – already on disk)
- For T3
  - set D to 19
  - if T3 had started before START CKPT, would have had to look before START CKPT for more actions to be undone



Summary

- UNDO logging
  - <T, X, u>: u is the old value of X
  - <T, X, u> to disk → X = new value to disk → … <COMMIT T> to disk
  - undo uncommitted transactions
- REDO logging
  - <T, X, v>: v is the new value of X
  - <T, X, v> to disk → …. <COMMIT T> to disk → X = new value to disk …
  - redo committed transactions
- UNDO/REDO logging
  - <T, X, u, v>: u is the old value of X and v is the new value of X
  - <T, X, u, v> to disk → X = new value to disk
  - No constraints on writing <COMMIT T> to disk
  - both: undo uncommitted and redo committed transactions
- Understand for each of these three
  - standard recovery
  - checkpointing, and
  - recovery with checkpointing