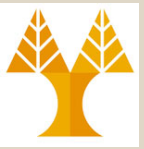
Department of Computer Science
University of Cyprus



EPL646 – Advanced Topics in Databases

Lecture 10

Crash Recovery: Undo, Redo, Undo/Redo Logging and Recovery

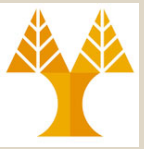
Chapter 17: Database Systems: The Complete Book
Garcia-Molina, Ullman, Widom, 2ED

Demetris Zeinalipour
http://www.cs.ucy.ac.cy/~dzeina/courses/epl646

Recovery: Outline

- Recovery: Definitions, Purpose, Failure Reasons, ACID Properties and Responsibilities
- **Three Types of Recovery**
  - **Undo** (uncommitted) Logging
  - **Redo** (committed) Logging
  - **Undo/Redo** Logging (not discussed)
- Checkpointing and Nonquiescent Checkpointing

Database Recovery
Ανάκαμψη σε Βάσεις Δεδομένων



- **Purpose of Database Recovery (Σκοπός Ανάκαμψης)**
  - To bring the database into the last **consistent state**, which existed **prior to the failure**.
  - To **preserve transaction properties** (**Atomicity**, Consistency, Isolation and **Durability**), especially the bold properties.
- **Example:**
  - A **system crashes** before a **fund transfer** transaction commits its execution,
  - Either **one** or **both** accounts may have an **incorrect value.**
  - Thus, the database must be **restored to the state** before the transaction modified **any** of the accounts.

Failure Reasons of Transactions (Λόγοι Σφάλματος Δοσοληψιών)

Type of Crash	Prevention
Wrong data entry	Constraints and Data cleaning
Disk crashes	Redundancy: e.g. RAID, archive
Fire, theft, bankruptcy…	Buy insurance, Change jobs…
System failures: e.g. power, OS, RAM	**Our focus: DATABASE RECOVERY**

Most frequent

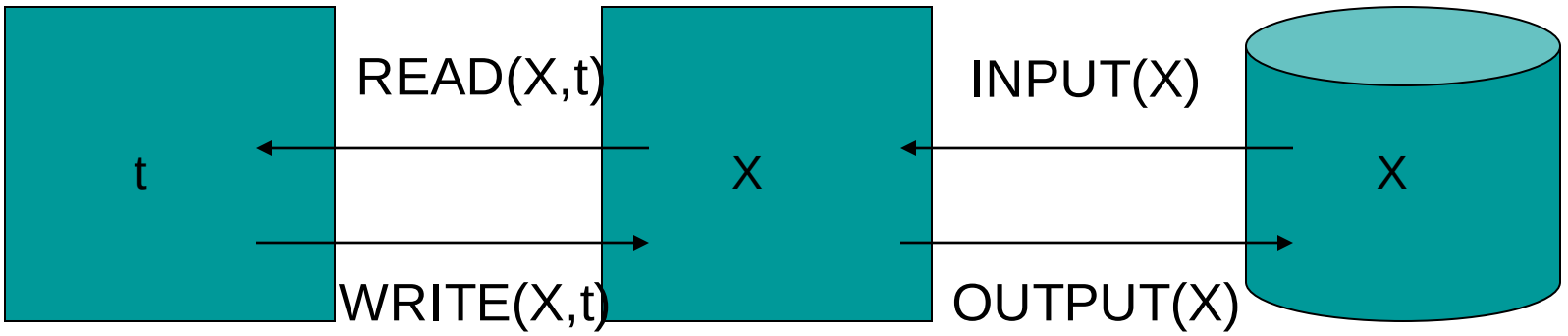
Review: The ACID properties (Επανάληψη: Οι ιδιότητες ACID)

- **Atomicity (Ατομικότητα):** All actions in the Xact happen, or none happen (Responsibility: **Recovery Manager**).
- **(Semantic) Consistency (Συνέπεια):** If each Xact is consistent, and the DB starts consistent, it ends up consistent (Responsibility: **User … using constraints**).
- **Isolation (Απομόνωση):** Execution of one Xact is isolated from that of other Xacts (Responsibility: **Concurrency Control Manager**).
- **Durability (Μονιμότητα):** If a Xact commits, its effects persist (Responsibility: **Recovery Manager**).

- The **Recovery Manager** guarantees **Atomicity** & **Durability**

System Model & Definitions (Μοντέλο Συστήματος & Ορισμοί)

- Assumption: the database is composed of reads/writes over some ***elements***
  - Usually 1 element = 1 block
  - Can be smaller (=1 record) or larger (=1 relation)
- Symbols Utilized
  - X database object
  - t local (program) variable



program (memory)

memory buffer (aka Buffer Manager)

Database disk

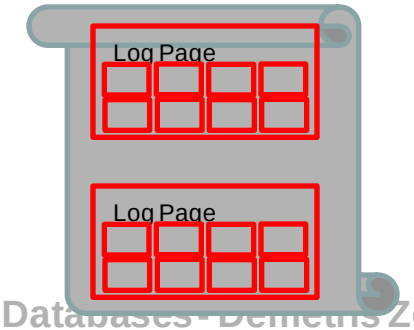
Example Used in these Slides

READ(A,t); t := t*2;WRITE(A,t); READ(B,t); t := t*2;WRITE(B,t)

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	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
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

System Failures and Logs

- Each transaction (program) has an ***internal state***
- When system crashes, internal state is lost
  - Don't know which parts executed and which didn't
- Remedy: use a **log**
  - A **sequential file** that keeps track of all **transaction operations** that **affect the values** of database **items**.
  - Log is maintained both on disk and buffer (will see next) where, how, when.



Transaction Log (Κατάστιχο Δοσοληψιών)

- An **append-only** file containing **log records**
- **Note:** multiple transactions run concurrently, log records are interleaved
- After **system crash**, use log to do either or both:
  - **Undo** updates of **uncommitted** xacts **(from end)**
  - **Redo** updates of **committed** xacts (that have not been OUTPUTTED to disk) **(from start)**
- Three kinds of logs: undo, redo, undo/redo

Recovery A: **Undo** Logging

Log records

- **<START T>**
  - transaction T has begun
- **<COMMIT T>**
  - T has committed
- **<ABORT T> (aka. Rollback)**
  - T has aborted
- **<T,X,v>**
  - T has updated element X, and its ***old*** value was v

Undo-Logging Example (Log-then-output, Commit later)

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				(now all log records appear on disk)		
Takes time **OUTPUT(A)**	16	16	16	16	8	
Takes time **OUTPUT(B)**	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG				(now COMMIT T appears on disk as well)		

old value

!!! CRASH => UNDO OUTPUT(A)

Undo-Logging Rules

The following two rules must be obeyed:

U1 (Log then Output) : If **T** modifies **X**, then **<T,X,v>** must be written to disk **BEFORE X** is output to disk

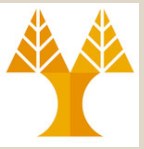
π.χ., Write to Log: <T,A,8> ***then*** *Flush Log* ***then***
Write to Disk: OUTPUT A

U2 (Commit Later): If **T** commits, then **<COMMIT T>** must be written to disk only after all changes by **T** are output to disk

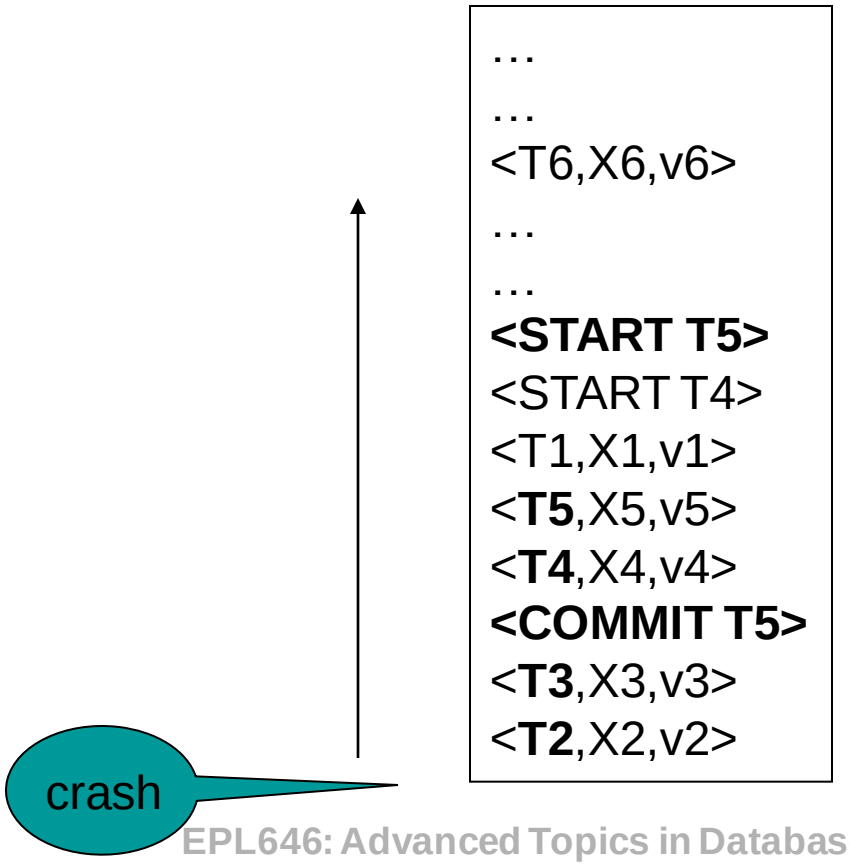
π.χ., Write to Disk: OUTPUT A ***then***
Write to Log: COMMIT A ***then*** *Flush Log*

– Hence: **OUTPUTs** are done ***early***, before the transaction commits

Recovery with Undo Log (Abort Uncommitted)



General Idea: Undo Uncommitted Transactions (see next slide for details)



Question1:
Which updates are undone ?

Question 2:
How far back do we need to read in the log ?

Recovery with Undo Log

After system's crash, run Recovery Manager

- **Step 1.** Process the Log from **END** and decide for each **transaction T** whether it is completed or not (how? See next slide)
  - <START T>….<COMMIT T>…. = COMPLETE
  - <START T>….<ABORT T>……. = COMPLETE
  - <START T>…… = INCOMPLETE
- **Step 2.** Undo all modifications by <u>incomplete</u> transactions (see next slide)

Recovery with Undo Log

How does the Recovery Manager classify Xacts?

- Read log **from the end**; cases:
  - <**COMMIT T>:** mark T as completed
  - **<ABORT T>:** mark T as completed
  - **<T,X,v>:** if (T is completed) then
    Ignore
    else // incomplete
    Write X=oldest(v) to disk (i.e. reset X to its initial value, higher in the log)
  - **<START T>:** ignore

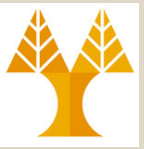
Crashing During Recovery with Undo Log

- **System Crash during Recovery with Undo Log. What happens?**
  - We can repeat all actions from scratch for a second time, no harm is done.
- **Why?**
  - All undo commands are *idempotent (ταυτοδύναμες)*
    - **T1(A); crash; T1(A); T2(A)**

    Generates the same result with:
    - **T1(A); T2(A)**

    because «**<Ti,A,v>** holds the previous value v of object A (wouldn't apply if v was holding the difference from previous value)

Recovery with Undo Log



When do we stop reading the log ?

- We cannot stop until we reach the **beginning of the log file**

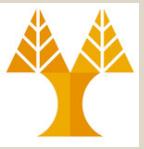
Why?

- Think about <START T1> on first line without <COMMIT|ABORT T1>

So Recovery is not very practical!

- Better idea: use checkpointing

Checkpointing (Σημείο Έλεγχου)

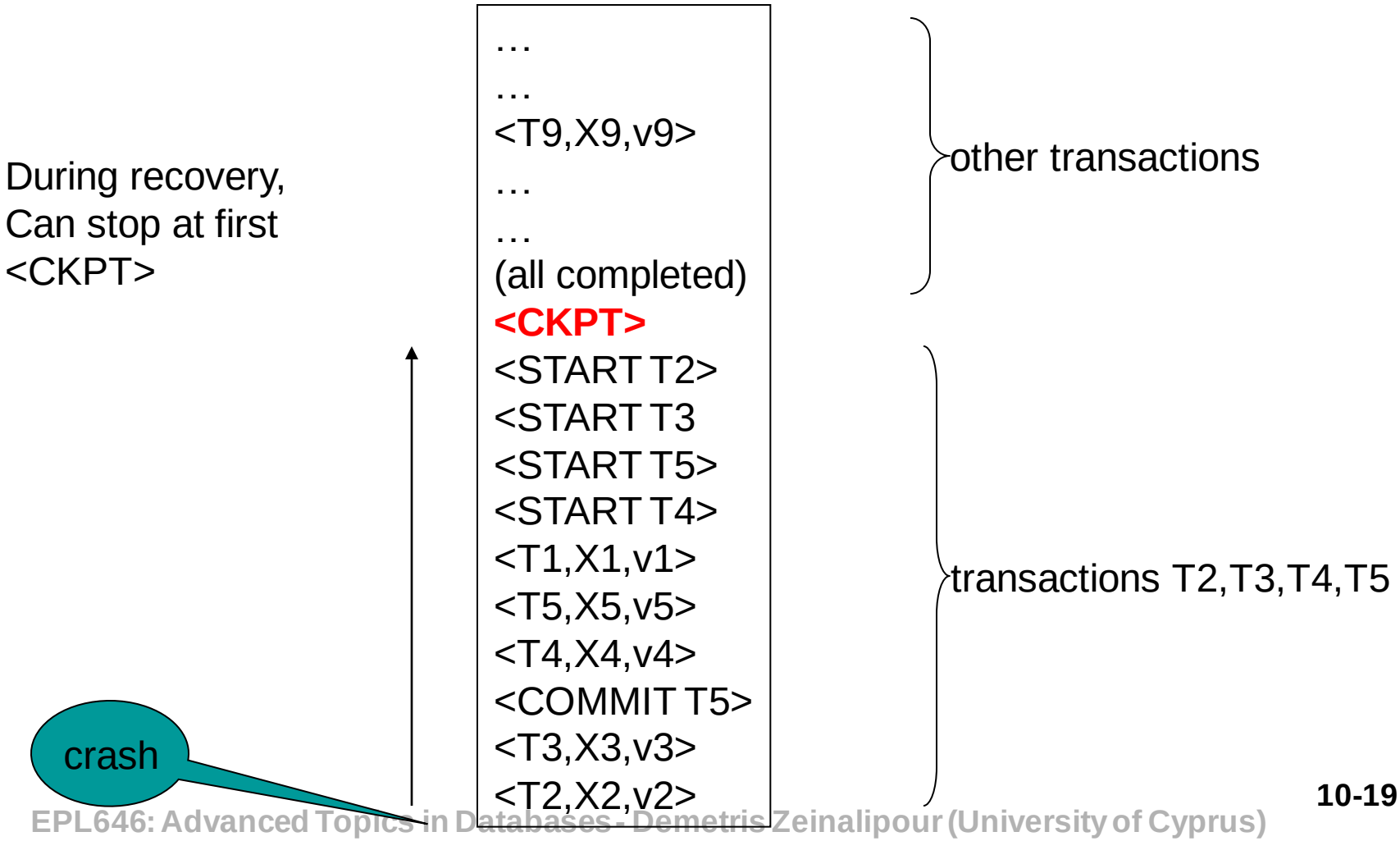


Idea: Checkpoint the database **periodically.**

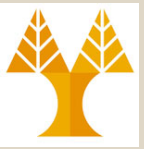
How?

- Stop accepting new transactions
- Wait until all current transactions complete
- Write a <CKPT> log record
- Resume transactions

Undo Recovery with Checkpointing

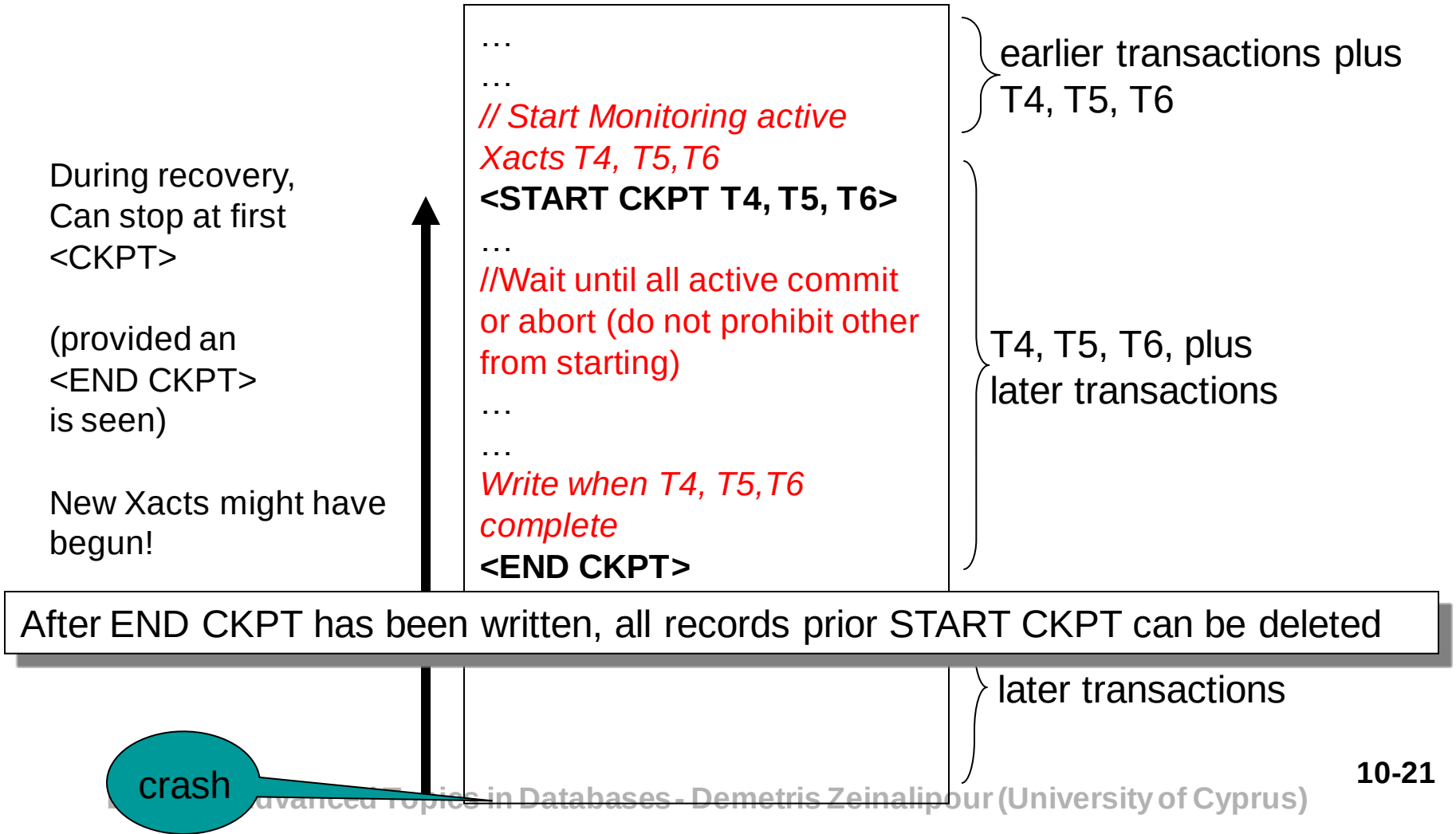


Nonquiescent Checkpointing (Μη-αδρανές Checkpointing)



- **Problem with checkpointing:** What happens with long running xacts ? => database freezes during checkpoint
- Would like to checkpoint while database is operational
- **Idea: Nonquiescent Checkpointing** (μη-αδρανές σημείο έλεγχου):
  - Write a **<START CKPT(T1,…,Tk)>** where **T1,…,Tk** are **all** active transactions
  - Continue normal operation
  - When all of **T1,…,Tk** have completed, write **<END CKPT>**

Undo Recovery with Nonquiescent Checkpointing



Recovery B: **Redo** Logging

Problems of UNDO Logging: It requires to **OUTPUT** the data before **COMMIT**. Thus, in order to offer **Strict Schedules** we must write all in-memory data to disk.

Log records

- <START T> = transaction T has begun
- <COMMIT T> = T has committed
- <ABORT T>= T has aborted
- **<T,X,v>= T has updated element X, and its *new* value is v**

Redo-Logging Example (Commit-then-Output)

IREAD: INPUT & READ

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
IREAD(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	**<T,A,16>**
IREAD(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,16>**
						<COMMIT T>
FLUSH LOG	(now all log records appear on disk). The next OUTPUT commands can in practice come much later					
OUTPUT(A)	16	16	16	16	8	**!!! CRASH =>**
OUTPUT(B)	16	16	16	16	16	**REDO COMMITTED not OUTPUTED!**

Other Xacts might start using A, B (in a "strict" way) without pulling them from disk (i.e., directly from buffer manager) . Not applicable in Undo Logging (where data needs to be first OUTPUT to disk (before COMMIT)

Redo-Logging Rule (Write-Ahead Rule)

R1 (Commit-then-Output): If **T** modifies **X**, then both **<T,X,v>** and **<COMMIT T>** must be written to disk before **X** is written to disk

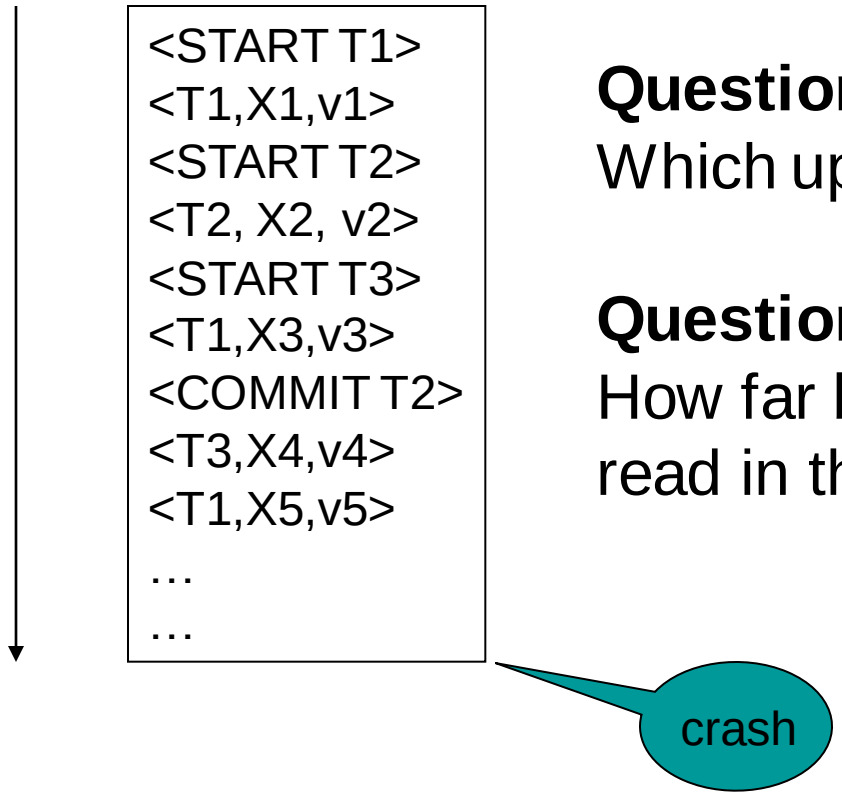
i.e., *Write to Log: COMMIT A*

Write to Disk: OUTPUT A

- Hence: OUTPUTs are done *late*

- ***This rule also known as Write-Ahead-Rule i.e., Write Ahead Log (WAL), i.e., writing log records ahead of actual data records.***

Recovery with Redo Log (Repeat Committed)



Question1:
Which updates are redone ?

Question 2:
How far back do we need to read in the log down ?

Recovery with Redo Log (Repeat Committed)

After system's crash, run recovery manager

- Step 1. Process the Log from **START** and decide for each **transaction T** whether it is completed or not (how? See next slide)
  - **<START T>….<COMMIT T>…. = COMPLETE**
  - **<START T>….<ABORT T>……. = COMPLETE**
  - **<START T>…… = INCOMPLETE**
- Step 2. Read **log from the beginning**, **redo all updates of *committed* transactions (not outputed)**
  - Do not repeat the uncommitted ones (let the application that initiated them worry about repeating them)

Recovery with Redo Log

How does the Recovery Manager classify Xacts?

- Read log from **end** :
  - <**COMMIT T>:** mark T as completed
  - **<ABORT T>:** mark T as completed
- Read log from the **start**; cases:
  - **<T,X,v>:** if (T is **incomplete**) then // opposite to UNDO
    Ignore
    else // complete & committed
    Write **X=newest(v)** to disk
  - **<START T>:** ignore
- For each incomplete **T** write an **<Abort T>** record to the end of the log and **flush the log**.

Nonquiescent Checkpointing with REDO Logging

- Write a <START CKPT(T1,…,Tk)> where **T1,…,Tk** are *all* active transactions
  - *Same with UNDO Checkpointing*
- **Flush to disk all blocks of committed transactions (*dirty blocks*), while continuing normal operation**
  - This solves the problem: DIRTY blocks go to disk
- When all blocks have been written, write <END CKPT>
  - *Same with UNDO Checkpointing*

Redo Recovery with Nonquiescent Checkpointing

Step 1: look for the last <END CKPT>

Step 2: find corresponding <START CKPT …>

…
<START T1>
…
<COMMIT T1>
…
<START T4>
…
<START CKPT **T4**, T5, T6>
…
All OUTPUTs of T1 are now known to be on disk
…
…
<END CKPT>
…
// now T4 committed in log but not OUTPUTed.
…
<START CKPT T9, T10>
…

crash

Step 3: redo from the earliest start of a transaction listed in <START CKPT …>, i.e. T4, T5, T6

(transactions committed earlier can be ignored, e.g., T1., as these were OUTPUTed by prior START..END block)