CS411
Database Systems

10: Transaction Management

Outline

- Transaction management
  - motivation & brief introduction
  - major issues
    - recovery
    - concurrency control
- Recovery

Users and DB Programs

- End users don't see the DB directly
  - are only vaguely aware of its design
  - may be aware of part of its contents
  - SQL is not a suitable end-user interface
- A single SQL query is not a sufficient unit of DB work
  - May need more than one query
  - May need to check constraints not enforced by the DBMS
  - May need to do calculations, realize "business rules", etc.

Users and DB Programs

- Ergo, a program is needed to carry out each unit of DB work
- End users interact with DB programs
  - Rather than SQL
  - May be many users simultaneously
    - Thus many simultaneous executions of these programs
  - Each user expects service and correct operation
    - A user should not have to wait forever
    - A user should not be affected by errors of others

Definition of "Transaction"

Definition: A transaction is the execution of a DB program.

- DB applications are designed as a set of transactions
- Typical transaction
  - starts with data from user or from another transaction
  - includes DB reads/writes
  - ends with display of data or form, or with request to start another transaction

Atomicity

- Transactions must be "atomic"
  - Their effect is all or none
  - DB must be consistent before and after the transaction executes (not necessarily during!)
- EITHER
  - a transaction executes fully and "commits" to all the changes it makes to the DB
  - OR it must be as though that transaction never executed at all

A Typical Transaction

- ***User view: "Transfer money from savings to checking"***
- Program:
  - read savings;
  - verify balance is adequate;
  - update savings balance;
  - read checking;
  - update checking balance;

"Commit" and "Abort"

- A transaction that only READs expects DB to be consistent, and cannot cause it to become otherwise.
- When a transaction which does any WRITE finishes, it must either
  - COMMIT: "I'm done and the DB is consistent again" OR
  - ABORT (ROLLBACK): "I'm done but I goofed: my changes must be undone."

Complications

- A DB may have many simultaneous users
  - simultaneous users implies simultaneous transactions implies simultaneous DB access
    - multiprogramming/multiprocessing
- Things can go wrong!
  - transactions can conflict with one another
  - programs may crash, OS may crash, disk may crash
    - company loses customer, gets sued, goes bankrupt, etc.

But DB Must Not Crash

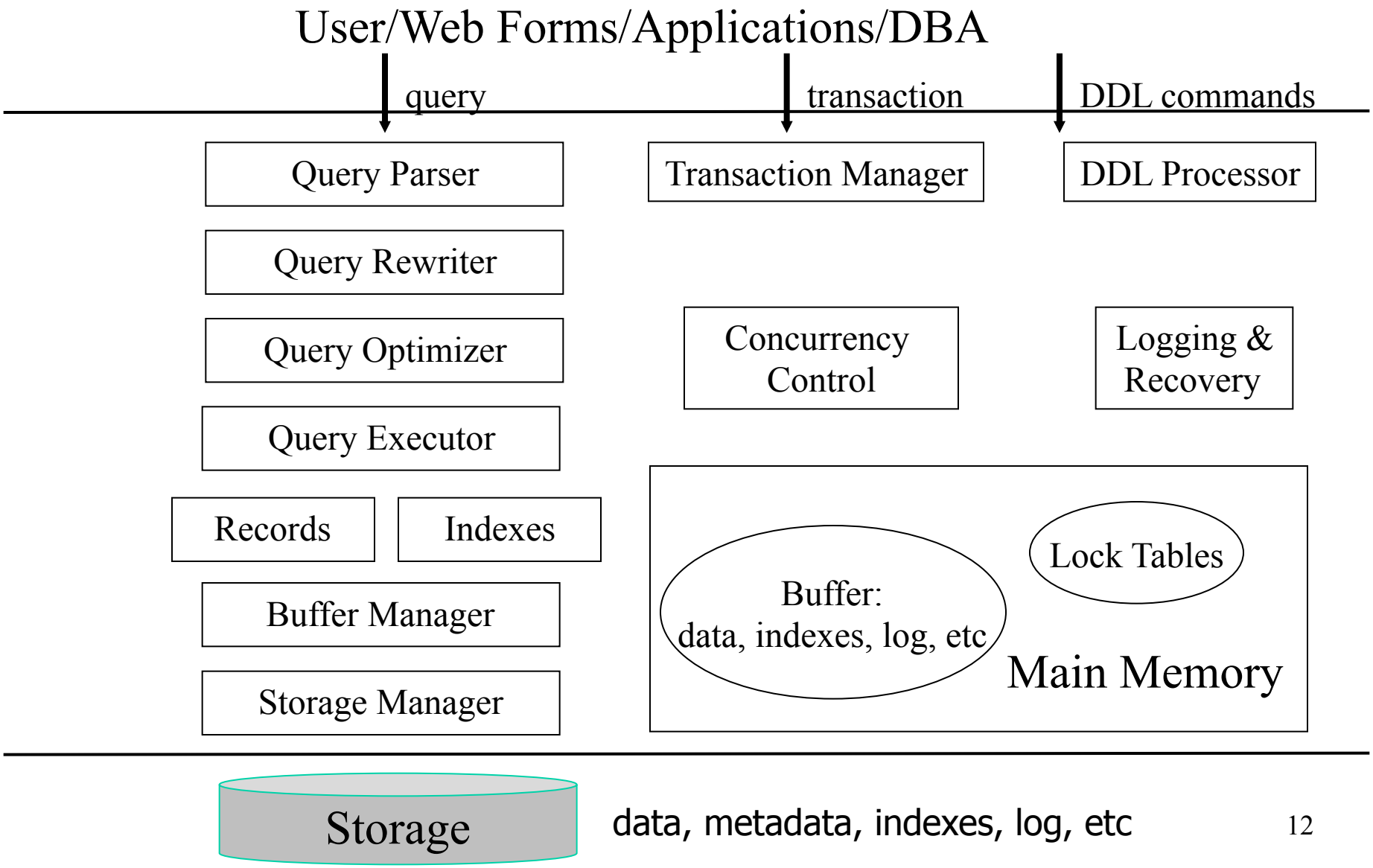
- Can't be allowed to become inconsistent
  - A DB that's 1% inaccurate is 100% unusable.
- Can't lose data
- Can't become unavailable

Can you name information processing systems that are more error tolerant?

Transaction Manager (or TP Monitor)

- Part of the DBMS
- Main duties:
  - Starts transactions
    - locate and start the right program
    - ensure timely, fair scheduling
  - Logs their activities
    - especially start/stop, writes, commits, aborts
  - Detects or avoids conflicts
  - Takes recovery actions

Big Picture: DBMS Architecture



What's on the Log File?

- Transaction starts/stops
- DB writes: "before" and/or "after" images of DB records
  - befores can be used to rollback an aborted transaction
  - afters can be used to redo a transaction (recovery from catastrophe)
- COMMITs and ABORTs

The log itself is as critical as the DB!

The Big TP Issues

- Recovery
  - Taking action to restore the DB to a consistent state
- Concurrency Control
  - Making sure simultaneous transactions don't interfere with one another

The ACID Properties

- Atomicity
- Consistency Preservation
- Isolation
- Durability

The ACID Properties: From Oracle Wiki

ACID

ACID refers to the basic properties of a database transaction: **A**tomicity, **C**onsistency, **I**solation, and **D**urability.

All Oracle database, Oracle RDB and InnoDB transactions comply with these properties. However, Oracle's Berkeley DB database is not ACID-compliant.

Atomicity

The entire sequence of actions must be either completed or aborted. The transaction cannot be partially successful.

Consistency

The transaction takes the resources from one consistent state to another.

Isolation

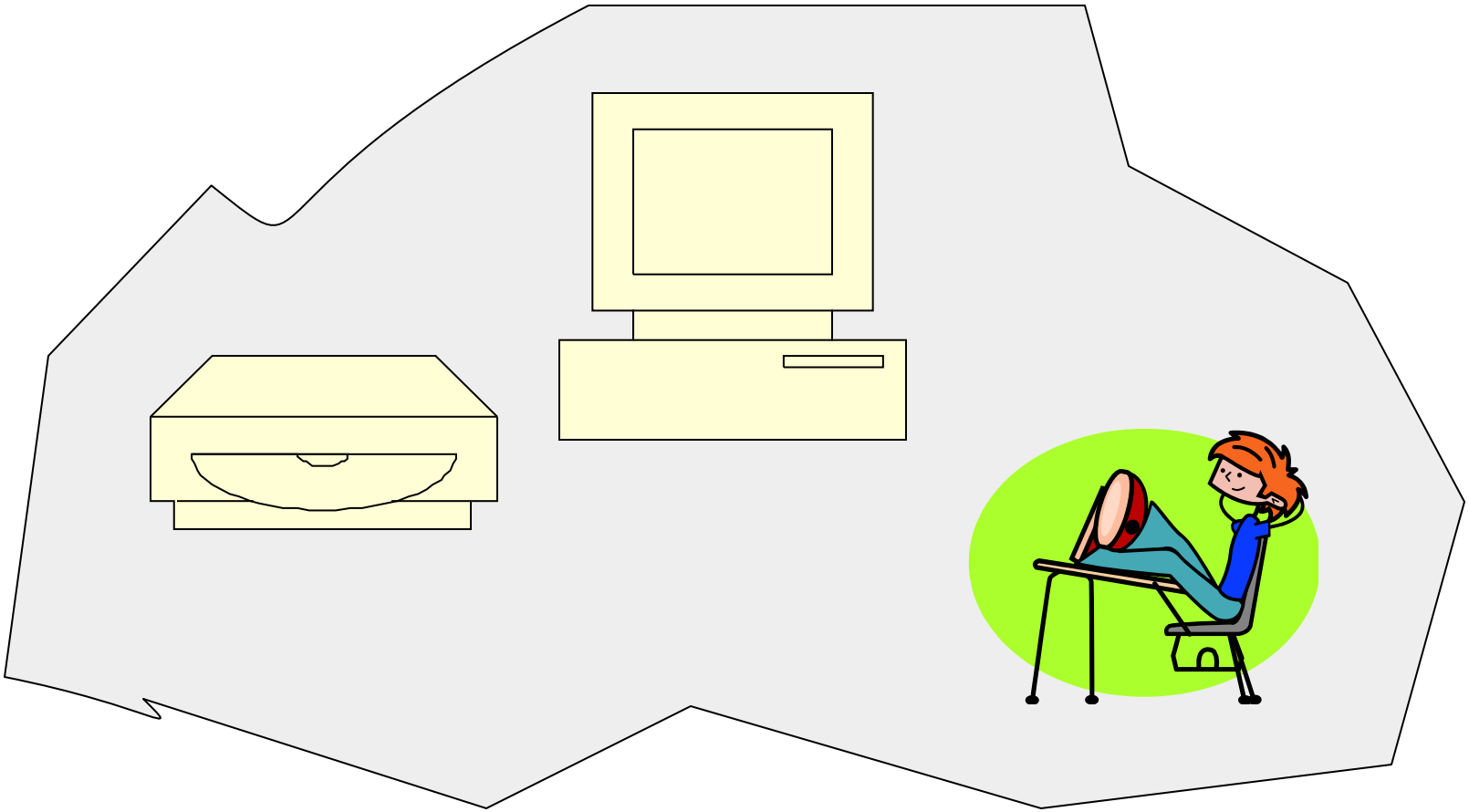
A transaction's effect is not visible to other transactions until the transaction is committed.

Durability

Changes made by the committed transaction are permanent and must survive system failure.

Recovery

Q: What Might Go Wrong?



System Failures

- Each transaction has *internal state*
- When system crashes, internal state is lost
  - Don't know which parts executed and which didn't
- Remedy: use a **log**
  - A file that records every single action of the transaction

Transactions

- Start Transaction
  - Oracle
    - by default, a new transaction is started after each COMMIT or ROLLBACK
  - MySQL
    - START TRANSACTION
- End Transaction
  - COMMIT or ROLLBACK

Transaction Abstraction

- Database is composed of ***elements***
  - Usually 1 element = 1 block
  - Can be smaller (=1 record) or larger (=1 relation)

- Each transaction reads/writes some elements

Correctness Principle

- There exists a notion of correctness for the database
  - Explicit constraints (e.g. foreign keys)
  - Implicit conditions (e.g. sum of sales = sum of invoices)

- ***Correctness principle***:

  Transaction will be programmed such that:

  if a transaction starts in a correct database state, it ends in a correct database state.

- Consequence: we only need to guarantee that transactions are ***atomic***, and run (as if) in isolation.

Primitive Operations of Transactions

- INPUT(X)
  - read element X to memory buffer
- READ(X,t)
  - copy element X to transaction local variable t
- WRITE(X,t)
  - copy transaction local variable t to element X
- OUTPUT(X)
  - write element X to disk

Example

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

The Log

- An append-only file containing log records
- Note: multiple transactions run concurrently, log records are interleaved
- After a system crash, use log to:
  - Redo some transaction that did commit.
  - Undo other transactions that did not commit.

Undo Logging

Undo Logging

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 *old* value was v

Undo-Logging Rules

U1: If T modifies X, then <T,X,v> entry must be written to log before X is written to disk

U2: If T commits, then <COMMIT T> entry must be written to log only after all changes by T are written to disk

- Hence: OUTPUTs are done *early* (before commit)

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>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>

Recovery with Undo Log

After system's crash, run recovery manager

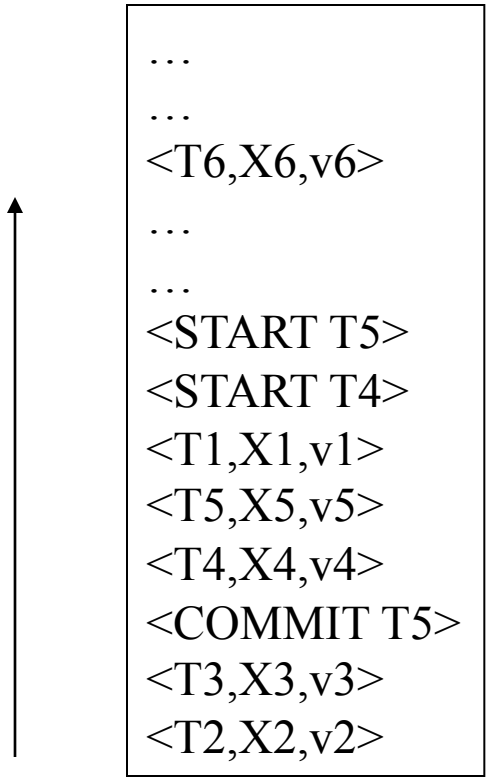
- Idea 1. Decide for each transaction T whether it is completed or not
  - <START T>….<COMMIT T>…. = yes
  - <START T>….<ABORT T>……. = yes
  - <START T>…… = no
- Idea 2. Undo all modifications by incomplete transactions

Recovery with Undo Log

Recovery manager:

- Read log from the end; cases:
  - <COMMIT T>: mark T as completed
  - <ABORT T>: mark T as completed
  - <T,X,v>: if T is not completed
    then write X=v to disk
    else ignore /* *committed or aborted xact.* */
  - <START T>: ignore

Recovery with Undo Log



Failure during recovery

- Note: all undo commands are *idempotent*
  - If we perform them a second time, no harm is done
  - E.g. if there is a system crash during recovery, simply restart recovery from scratch

When do we stop?

When do we stop reading the log ?

- We cannot stop until we reach the beginning of the log file
- This is impractical
- Better idea: use checkpointing

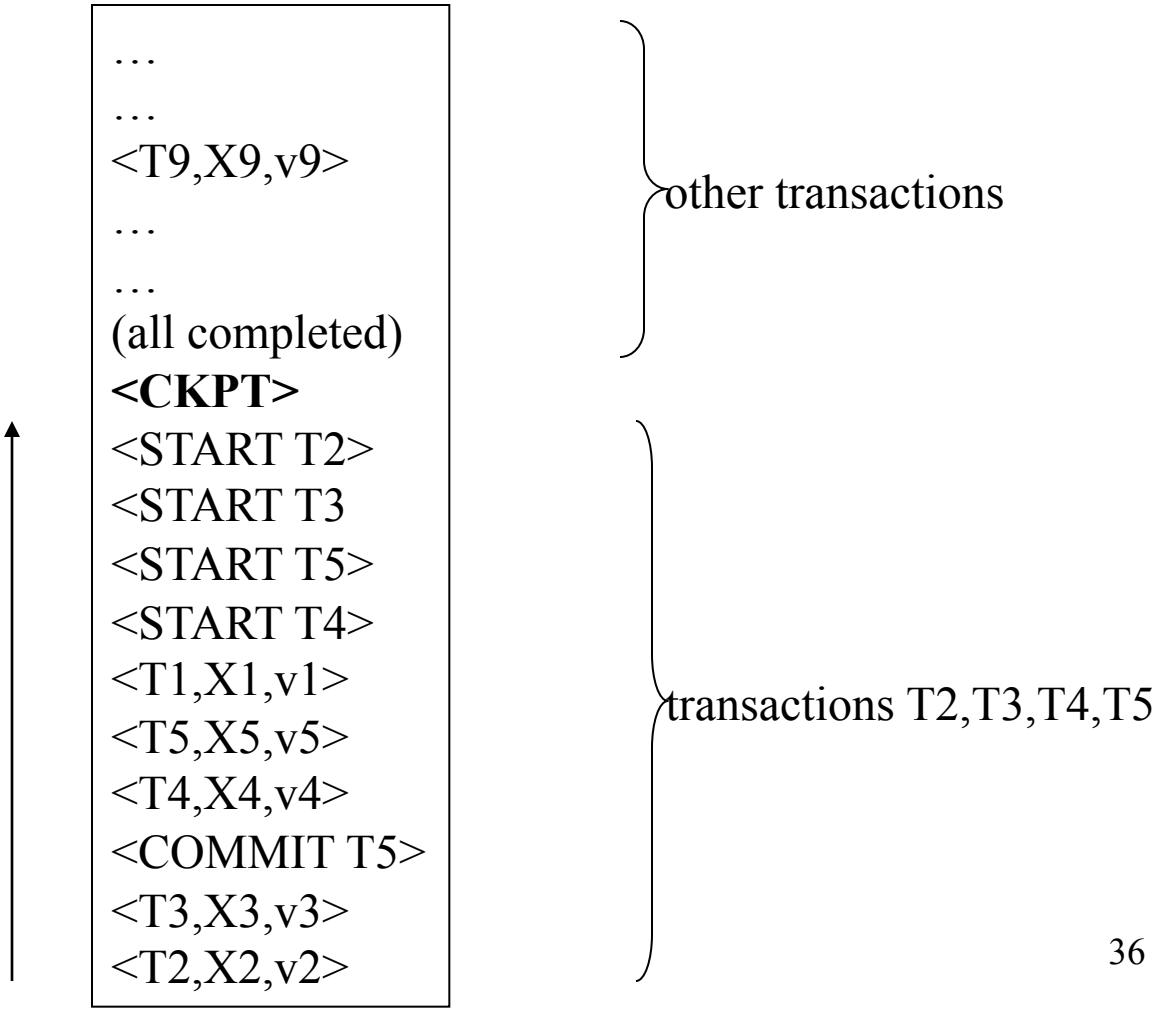
Checkpointing

Checkpoint the database periodically

- Stop accepting new transactions
- Wait until all curent transactions **complete**
- Flush log to disk
- Write a <CKPT> log record, flush
- Resume transactions

Undo Recovery with Checkpointing

During recovery,
Can stop at first
<CKPT>



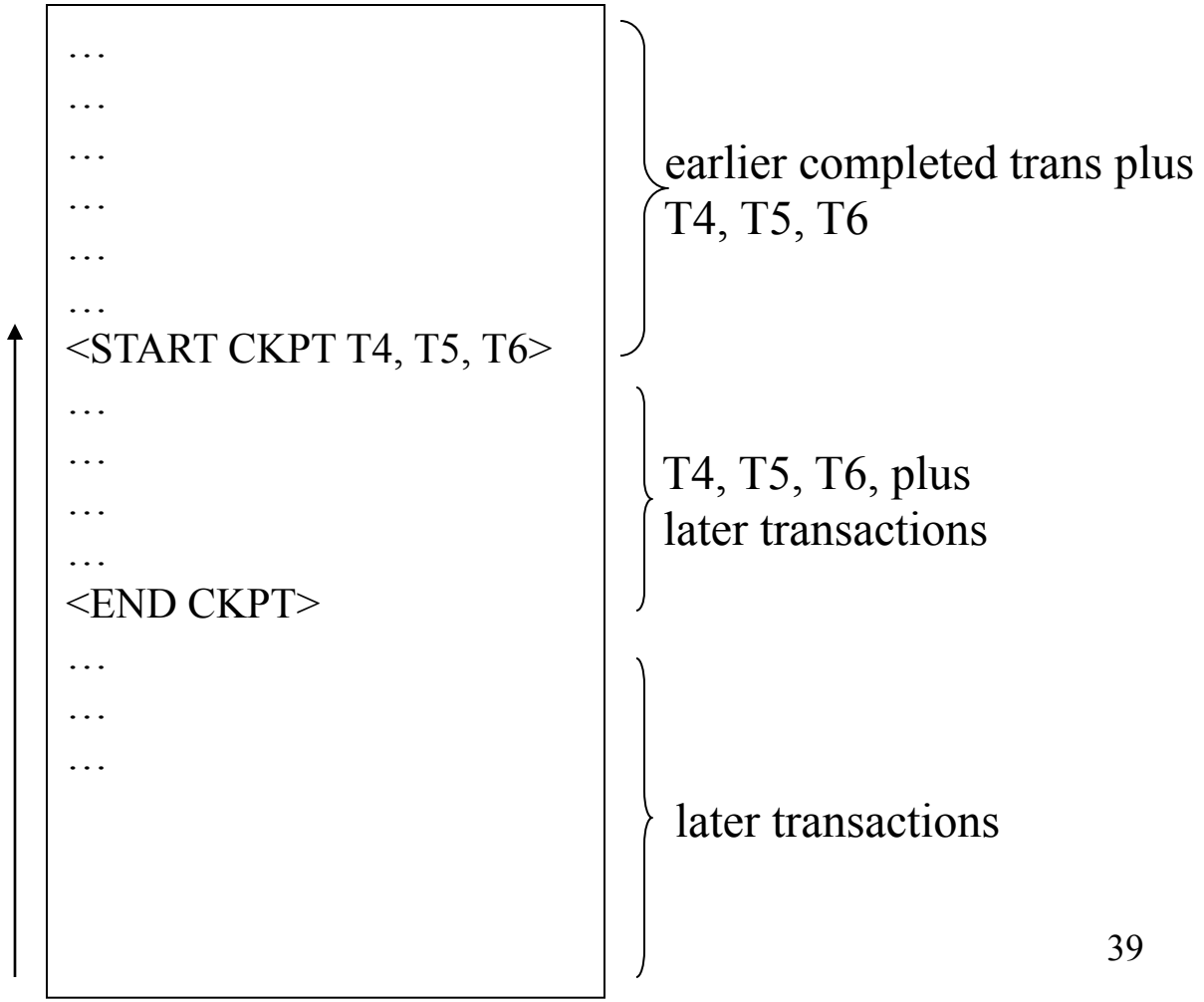
Nonquiescent Checkpointing

- Problem with checkpointing: database freezes during checkpoint
- Would like to checkpoint while database is operational
- =nonquiescent checkpointing

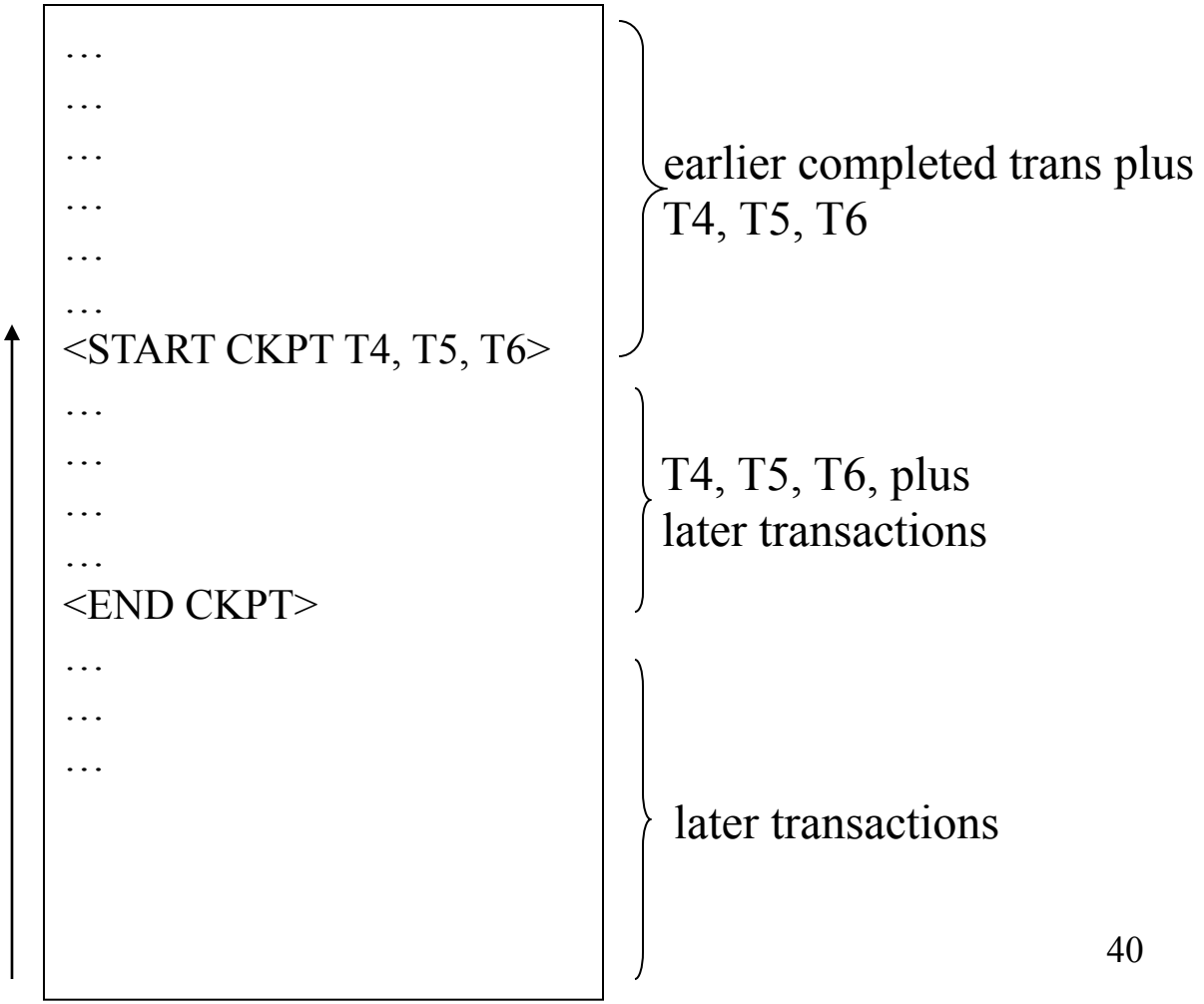
Nonquiescent Checkpointing

- 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



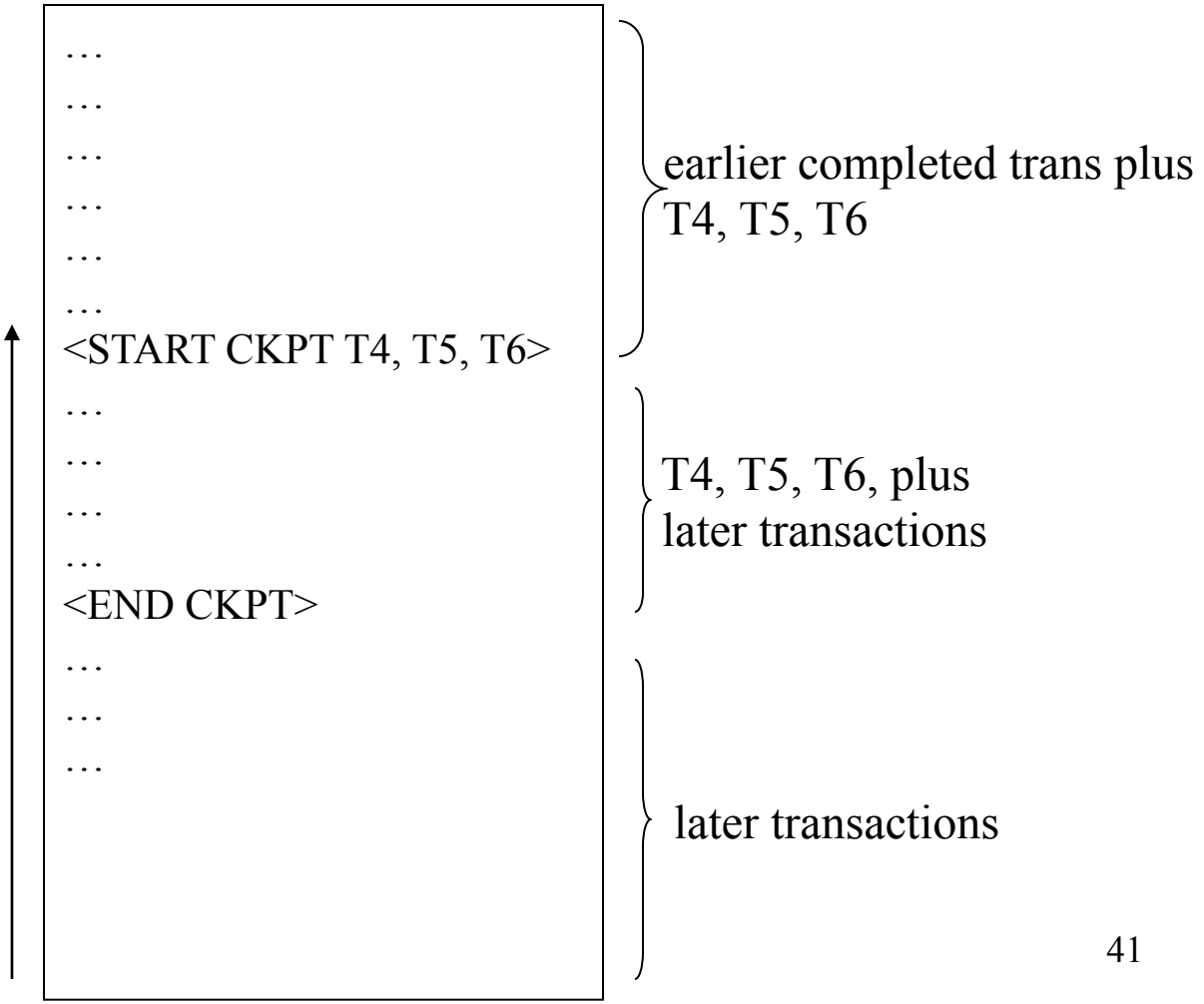
Undo Recovery with Nonquiescent Checkpointing



- Crash here -------->
- Undo any T started after <START CKPT>

Undo Recovery with Nonquiescent Checkpointing

- Crash here -------->
- Go to previous <START CKPT> (which has an <END CKPT>



Redo Logging

Redo Logging

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 Rules

R1: If T modifies X, then both <T,X,v> and <COMMIT T> must be written to disk before X is written to disk

- Hence: OUTPUTs are done *late* (after commit)

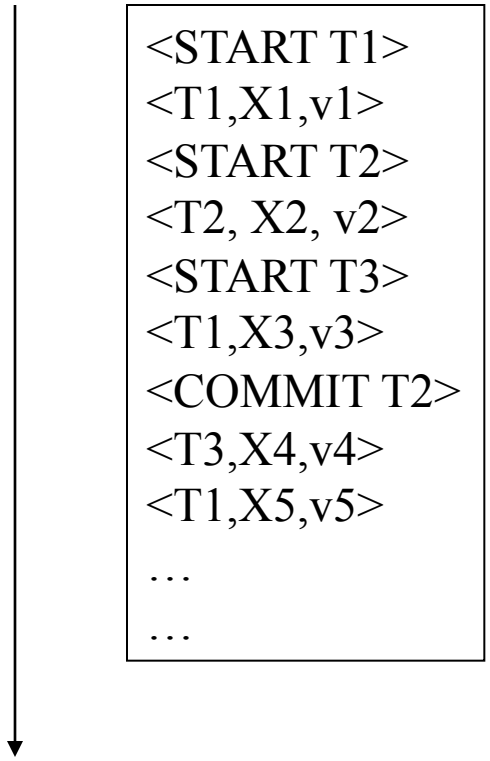
Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
REAT(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Recovery with Redo Log

After system's crash, run recovery manager

- Step 1. Decide for each transaction T whether it is completed or not
  - <START T>….<COMMIT T>…. = yes
  - <START T>….<ABORT T>……. = yes
  - <START T>…… = no
- Step 2. Read log from the *beginning*, redo all updates of *committed* transactions

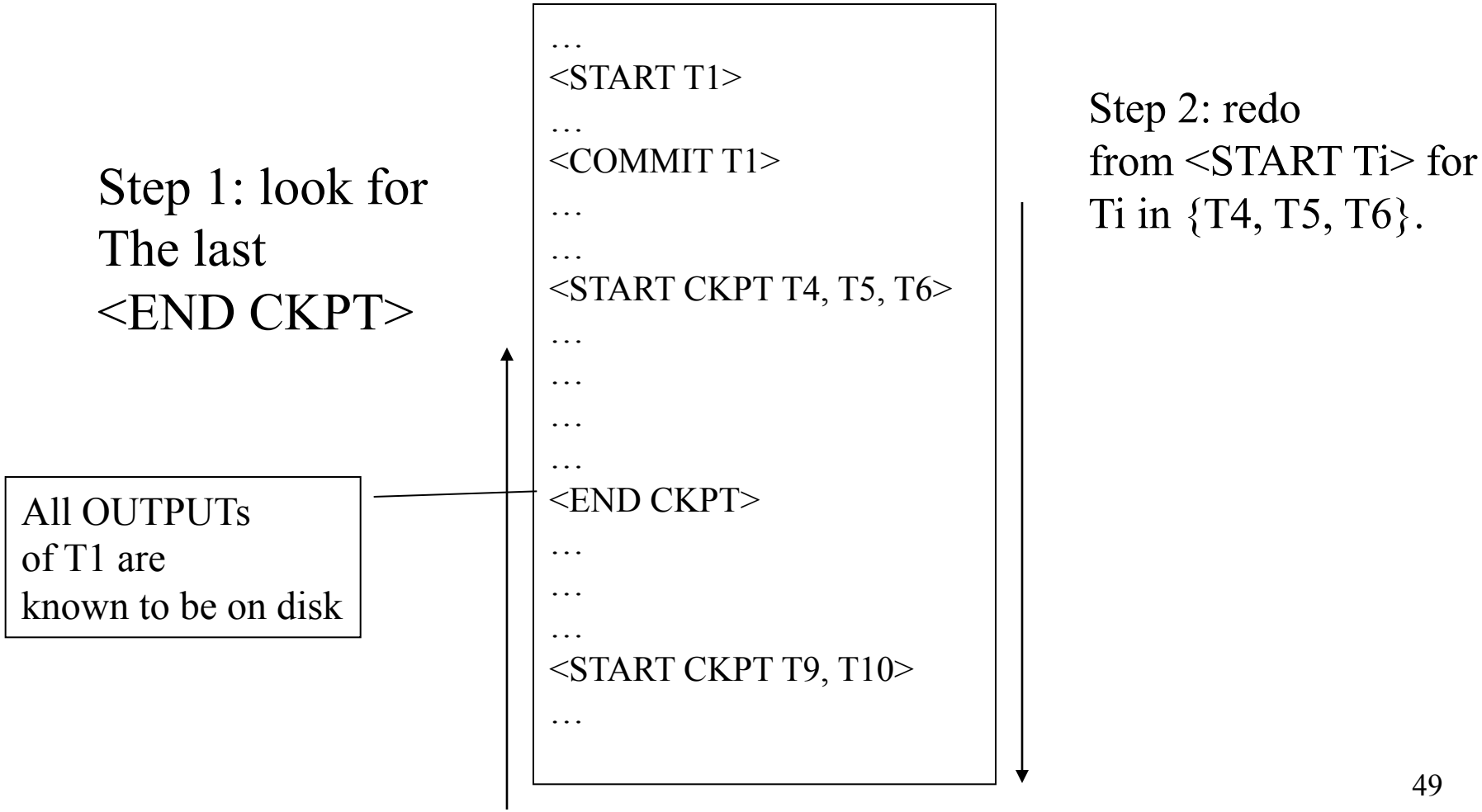
Recovery with Redo Log



Nonquiescent Checkpointing

- Write a <START CKPT(T1,…,Tk>
  where T1,…,Tk are all active transactions
- Flush to disk all blocks of *committed* transactions (*dirty blocks*), while continuing normal operation
- When all blocks have been written, write <END CKPT>

Redo Recovery with Nonquiescent Checkpointing



Comparison Undo/Redo

- Undo logging:
  - OUTPUT must be done early
  - If <COMMIT T> is seen, T definitely has written all its data to disk (hence, don't need to undo)
- Redo logging
  - OUTPUT must be done late
  - If <COMMIT T> is not seen, T definitely has not written any of its data to disk (hence there is not dirty data on disk)
- Would like more flexibility on when to OUTPUT: undo/redo logging (next)