

# Recovery

# Recovery

## Types of failures

- Wrong data entry
  - Prevent by having constraints in the database
  - Fix with data cleaning
- Disk crashes
  - Prevent by using redundancy (RAID, archive)
  - Fix by using archives
- Fire, theft, bankruptcy...
  - Buy insurance, change profession...
- System failures: *most frequent* (e.g. power)
  - Use recovery

- Accounts(id, checking, saving)

- START TRANSACTION

- UPDATE Accounts

SET checking = checking - 20

WHERE id = 123

- UPDATE Accounts

SET saving = saving + 20

Where id = 123

- COMMIT

- SELECT (checking + saving)

FROM Accounts

WHERE id = 123

# 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

- In ad-hoc SQL
  - each command = 1 transaction
- In embedded SQL (say inside a Python program)
  - Transaction starts = first SQL command issued
  - Transaction ends =
    - COMMIT
    - ROLLBACK (=abort)

# Transactions

- Assumption: the database is composed of *elements*
  - Usually 1 element = 1 block
  - Can be smaller (=1 record) or larger (=1 relation)
- Assumption: each transaction reads/writes some elements

# 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

INPUT(C); READ(C,t);  $t := t - 20$ ; WRITE(C,t); OUTPUT(C)

INPUT(S); READ(S,t);  $t := t + 20$ ; WRITE(S,t); OUTPUT(S)

Action	t	Mem C	Mem S	Disk C	Disk S
INPUT(C)		50		50	100
READ(C,t)	50	50		50	100
$t := t - 20$	30	50		50	100
WRITE(C,t)	30	30		50	100
INPUT(S)	30	30	100	50	100
READ(S,t)	100	30	100	50	100
$t := t + 20$	120	30	100	50	100
WRITE(S,t)	120	30	120	50	100
OUTPUT(C)	120	30	120	30	100
OUTPUT(S)	120	30	120	30	120

# 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 clean up transactions that haven't committed

# Undo Logging

## Log records

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

# Example

```
...
...
<T6,X6,v6>
...
...
<START T5>
<START T4>
<T1,X1,v1>
<T5,X5,v5>
<T4,X4,v4>
<COMMIT T5>
<T3,X3,v3>
<T2,X2,v2>
```

# Undo-Logging Rules

- U1: If T modifies X, then  $\langle T, X, v \rangle$  must be written to disk before X is written to disk
- U2: If T commits, then  $\langle \text{COMMIT } T \rangle$  must be written to disk only after all changes by T are written to disk
- Hence: OUTPUTs are done *early*

# Key Idea

- I will change values in this transaction
- To be safe, I will save the OLD values in log BEFORE I change any values in the table
- Once I'm done changing, I will say “I'm done” by putting a record <COMMIT T> into log
- Then you don't have to worry any more about this transaction

# Example

INPUT(C); READ(C,t); t := t - 20; WRITE(C,t); OUTPUT(C)

INPUT(S); READ(S,t); t := t + 20; WRITE(S,t); OUTPUT(S)

Action	t	Mem C	Mem S	Disk C	Disk S	Log
						<START T>
INPUT(C)		50		50	100	
READ(C,t)	50	50		50	100	
t:=t-20	30	50		50	100	
WRITE(C,t)	30	30		50	100	<T,C,50>
INPUT(S)	30	30	100	50	100	
READ(S,t)	100	30	100	50	100	
t:=t+20	120	30	100	50	100	
WRITE(S,t)	120	30	120	50	100	<T,S,100>
OUTPUT(C)	120	30	120	30	100	
OUTPUT(S)	120	30	120	30	120	
						<COMMIT T>

# Recovery with Undo Log

For the following discussion, will assume that transactions do not abort

Will discuss the case of abort later

# Recovery with Undo Log

After system's crash, run recovery manager

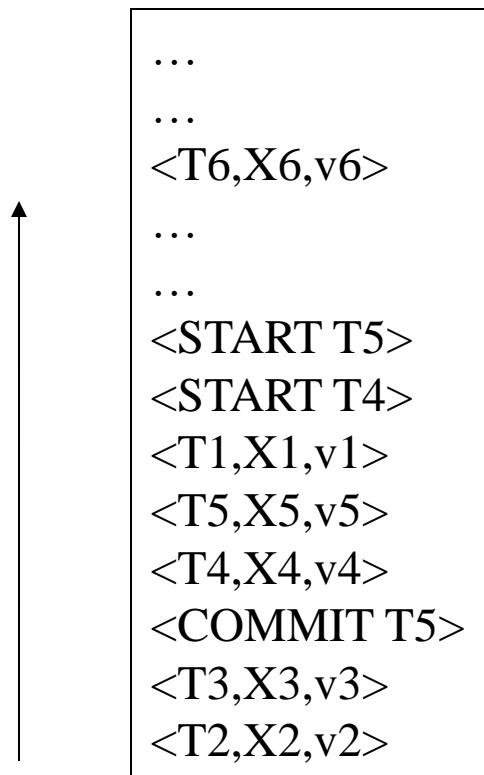
- Decide for each transaction T whether it is completed or not
  - $\langle \text{START T} \rangle \dots \langle \text{COMMIT T} \rangle \dots$  = yes
  - $\langle \text{START T} \rangle \dots \dots \dots \dots \dots$  = no
- Undo all modifications by incompletely completed transactions

# Recovery with Undo Log

Recovery manager:

- Read log from the end; cases:
  - $\langle \text{COMMIT } T \rangle$ : mark  $T$  as completed
  - $\langle T, X, v \rangle$ : if  $T$  is not completed
    - then write  $X=v$  to disk
    - else ignore
  - $\langle \text{START } T \rangle$ : ignore

# Recovery with Undo Log



# Recovery with Undo Log

- 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

# Recovery with Undo Log

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

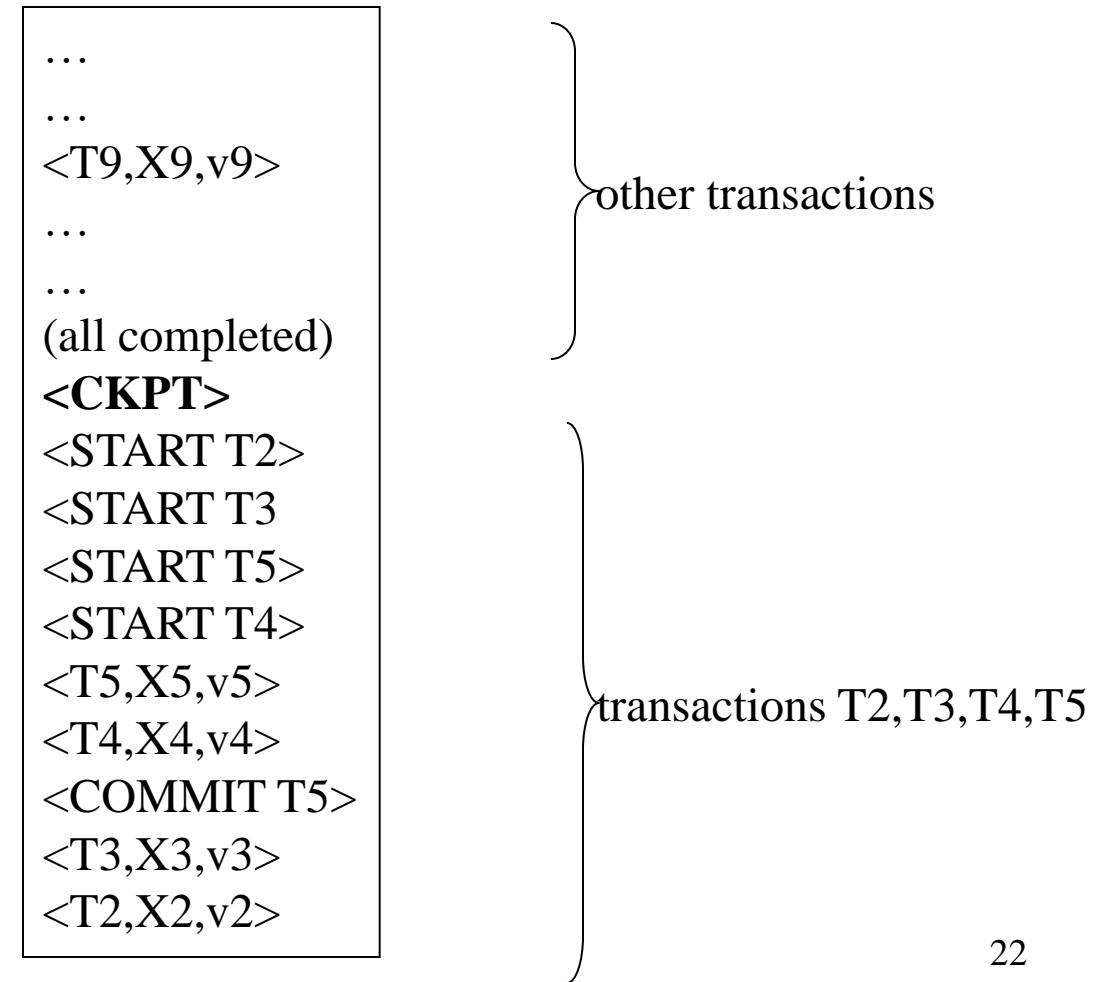
# Checkpointing

Checkpoint the database periodically

- Stop accepting new transactions
- Wait until all current 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

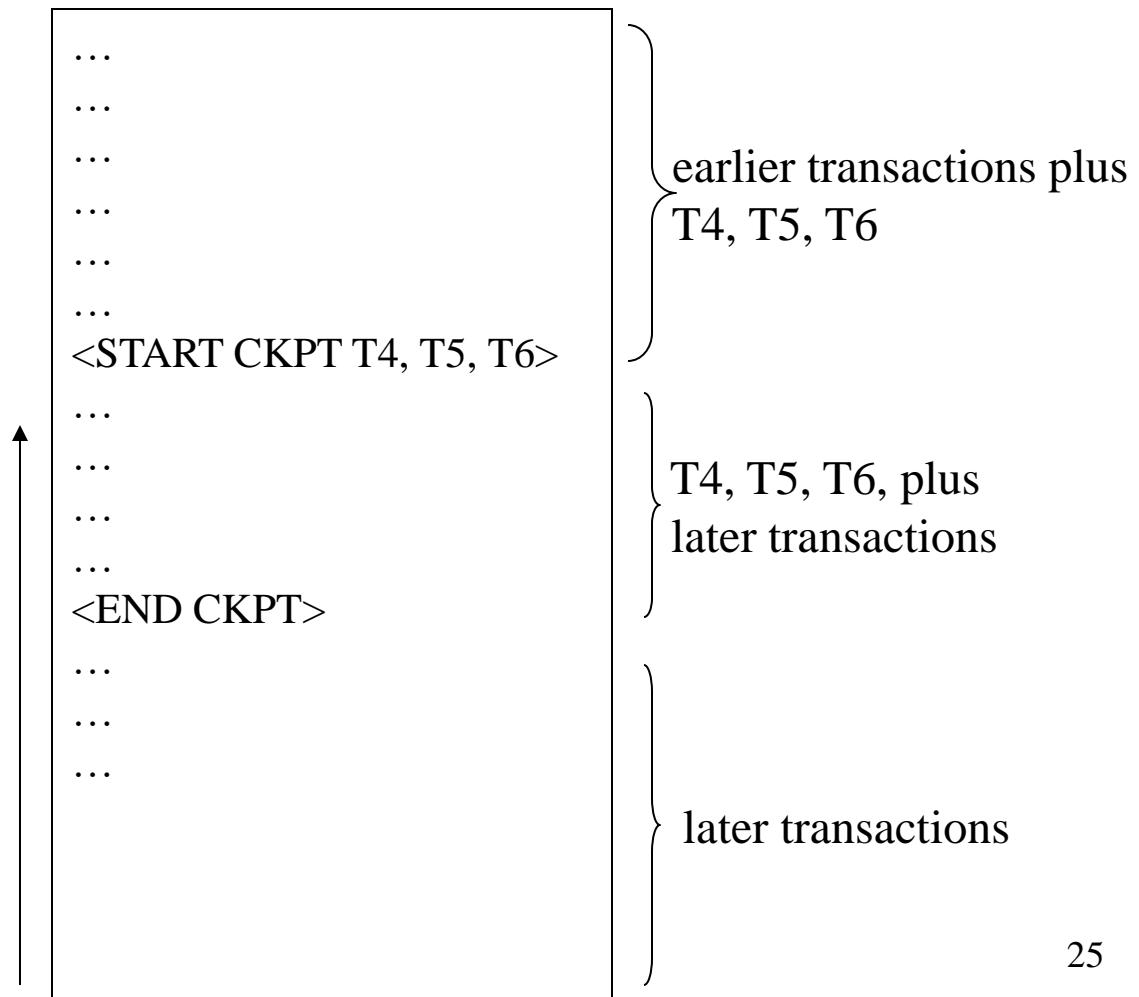
- Problem with checkpointing: database freezes during checkpoint
- Would like to checkpoint while database is operational
- =nonquiescent checkpointing

# Nonquiescent Checkpointing

- Write a  $\langle \text{START CKPT}(T_1, \dots, T_k) \rangle$  where  $T_1, \dots, T_k$  are all active transactions
  - All other transactions have committed
- Continue normal operation
- When all of  $T_1, \dots, T_k$  have committed or aborted, write  $\langle \text{END CKPT} \rangle$

# Undo Recovery with Nonquiescent Checkpointing

During recovery,  
Can stop at first  
 $\langle\text{CKPT}\rangle$



Q: why do we need  
 $\langle\text{END CKPT}\rangle$  ?

# A Transaction that Aborts

- Must undo its effect using the log even if there is no crash
- In effect do crash recovery for just that transaction, before it can release the locks

# Redo Logging

## Log records

- $\langle \text{START } T \rangle$  = transaction T has begun
- $\langle \text{COMMIT } T \rangle$  = T has committed
- $\langle \text{ABORT } T \rangle$  = T has aborted
- $\langle T, X, v \rangle$  = T has updated element X, and its new value is v

# Redo-Logging Rules

R1: If T modifies X, then both  $\langle T, X, v \rangle$  and  $\langle \text{COMMIT } T \rangle$  must be written to disk before **X** is written to disk

- Hence: OUTPUTs are done *late*

# Key Idea

- In this transaction, I PROMISE to change values in certain ways. Toward this goal, I will record the NEW values in log, and end with a <COMMIT T> in log
- Only AFTER this would I start changing values.

# Example

INPUT(C); READ(C,t); t := t - 20; WRITE(C,t); OUTPUT(C)

INPUT(S); READ(S,t); t := t + 20; WRITE(S,t); OUTPUT(S)

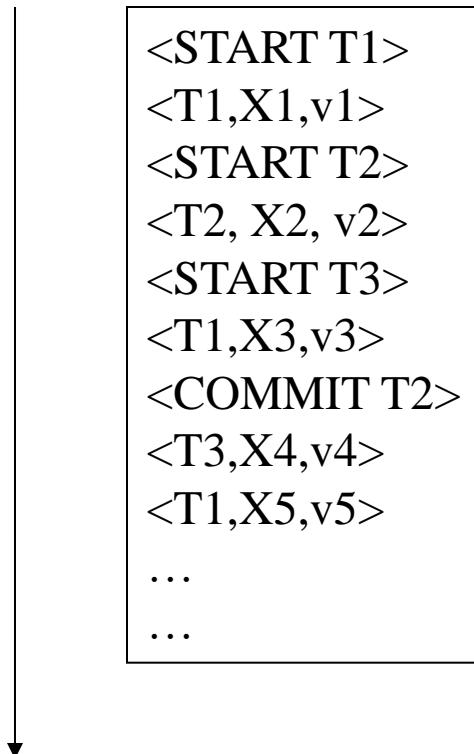
Action	t	Mem C	Mem S	Disk C	Disk S	Log
						<START T>
INPUT(C)		50		50	100	
READ(C,t)	50	50		50	100	
t:=t-20	30	50		50	100	
WRITE(C,t)	30	30		50	100	<T,C,30>
INPUT(S)	30	30	100	50	100	
READ(S,t)	100	30	100	50	100	
t:=t+20	120	30	100	50	100	
WRITE(S,t)	120	30	120	50	100	<T,S,120>
						<COMMIT T>
OUTPUT(C)	120	30	120	30	100	
OUTPUT(S)	120	30	120	30	120	

# Recovery with Redo Log

After system's crash, run recovery manager

- Decide for each transaction T whether it is completed or not
  - $\langle \text{START T} \rangle \dots \langle \text{COMMIT T} \rangle \dots$  = yes
  - $\langle \text{START T} \rangle \dots \dots \dots$  = no
- Read log from **the beginning**, redo all updates of committed transactions

# Recovery with Redo Log



# Nonquiescent Checkpointing

- Write a  $\langle \text{START CKPT}(T_1, \dots, T_k) \rangle$  where  $T_1, \dots, T_k$  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  $\langle \text{END CKPT} \rangle$

# Redo Recovery with Nonquiescent Checkpointing

Step 1: look for  
The last  
`<END CKPT>`

All OUTPUTs  
of T1 are  
known to be on disk

```
...  
<START T1>  
...  
<COMMIT T1>  
...  
...  
<START CKPT T4, T5, T6>  
...  
...  
...  
...  
<END CKPT>  
...  
...  
...  
...  
<START CKPT T9, T10>  
...
```

Step 2: redo  
all committed  
transactions that  
are listed in  
`<start ckpt ...>` and  
transactions starting  
after this `<start ckpt>`  
record

# Comparison Undo/Redo

- Undo logging:
  - OUTPUT must be done early
  - If  $\langle\text{COMMIT T}\rangle$  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  $\langle\text{COMMIT T}\rangle$  is not seen, T definitely has not written any of its data to disk (hence there is no dirty data on disk)
- Would like more flexibility on when to OUTPUT: undo/redo logging (next)

# Undo/Redo Logging

Log records, only one change

- $\langle T, X, u, v \rangle =$  T has updated element X, its *old* value was u, and its *new* value is v

# Undo/Redo-Logging Rule

UR1: If T modifies X, then  $\langle T, X, u, v \rangle$  must be written to disk before X is written to disk

Note: we are free to OUTPUT early or late  
(I.e. before or after  $\langle \text{COMMIT } T \rangle$ )

# Example

INPUT(C); READ(C,t); t := t - 20; WRITE(C,t); OUTPUT(C)

INPUT(S); READ(S,t); t := t + 20; WRITE(S,t); OUTPUT(S)

Action	t	Mem C	Mem S	Disk C	Disk S	Log
						<START T>
INPUT(C)		50		50	100	
READ(C,t)	50	50		50	100	
t:=t-20	30	50		50	100	
WRITE(C,t)	30	30		50	100	<T,C,50,30>
INPUT(S)	30	30	100	50	100	
READ(S,t)	100	30	100	50	100	
t:=t+20	120	30	100	50	100	
WRITE(S,t)	120	30	120	50	100	<T,S,100,120>
OUTPUT(C)	120	30	120	30	100	
						<COMMIT T>
OUTPUT(S)	120	30	120	30	120	

# Recovery with Undo/Redo Log

After system's crash, run recovery manager

- Redo all committed transaction, top-down
- Undo all uncommitted transactions, bottom-up