



PA152: Efficient Use of DB

# 11. Failure Recovery

Vlastislav Dohnal



# Contents

- Overview of integrity
- Transactions
- Logging in DBMS

# Integrity or correctness of data

- Would like data to be “accurate” or “correct” at all times

Employee

Name	Age
Newman	52
Altman	3421
Freeman	1

# Integrity or correctness of data

## ■ Integrity constraints

- Main approach to consistency of DB
- Predicates that data must satisfy

## ■ Examples:

- $\text{Domain}(x) = \{\text{red, blue, green}\}$
- $x$  is a key of relation  $R$
- A valid value for attribute  $x$  of  $R$  (foreign key)
- Functional dependency:  $x \rightarrow y$

# Integrity or correctness of data

- Consistent state
  - satisfies all constraints
- Consistent DB
  - DB in consistent state

# Limits of integrity constraints

- May not capture “full correctness”
- Examples: (Transaction constraints)
  - No employee should make more than twice the average salary.
  - Student scholarship may not exceed 30k per month in total.
  - When a bank account is deleted, balance = 0

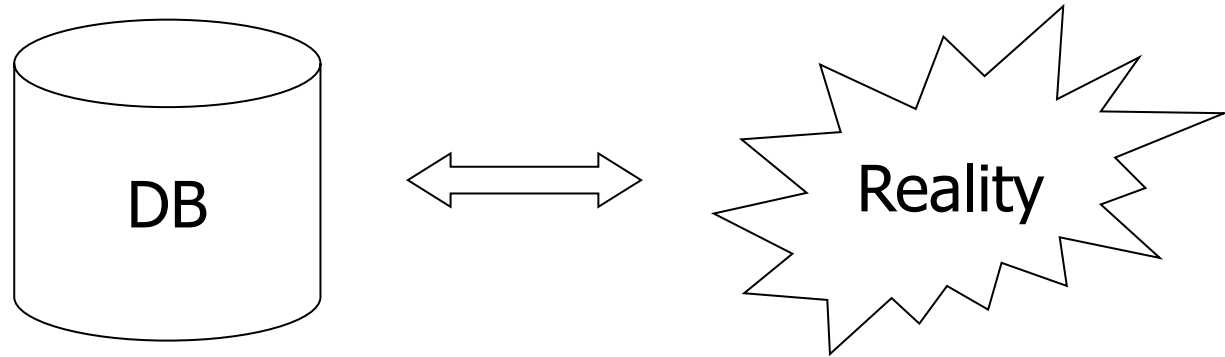
# Limits of integrity constraints

- Some could be “emulated” by simple constraints
  - Deletion of account replaced with deletion flag

account	acc.no.	...	balance	deleted
---------	---------	-----	---------	---------

# Limits of integrity constraints

- Database should reflect real world.



- Continue with constraints
  - even though some part of “reality” cannot be defined as constraint or DB does not mirror reality
- Observation
  - DB cannot always be consistent.



# Example of inconsistent state

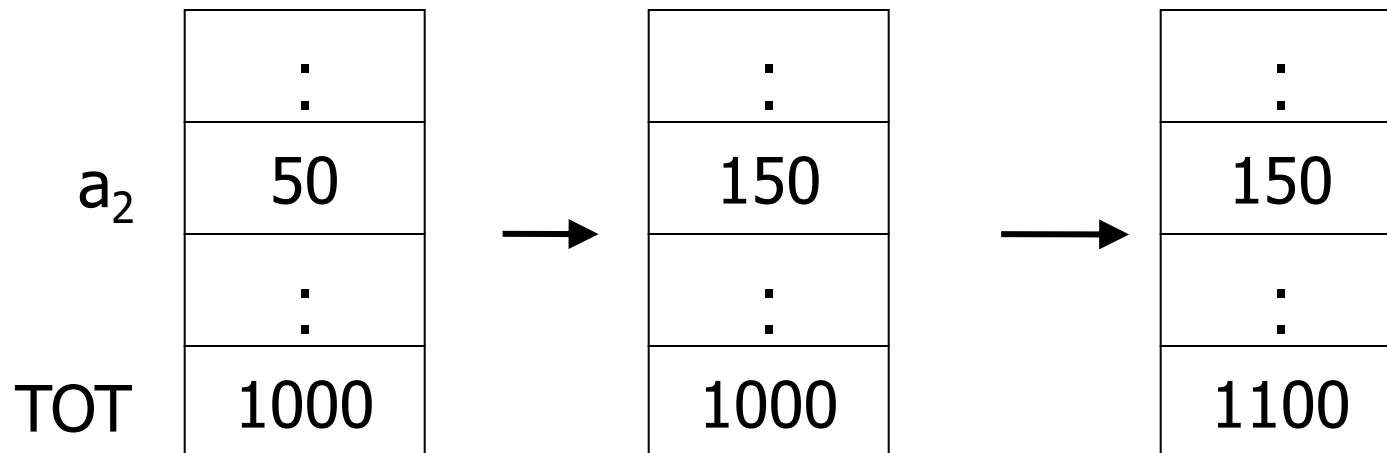
- Constraint example:

- $a_1 + a_2 + \dots + a_n = \text{TOT}$

- Depositing 100 CZK to account  $a_2$

- $a_2 \leftarrow a_2 + 100$

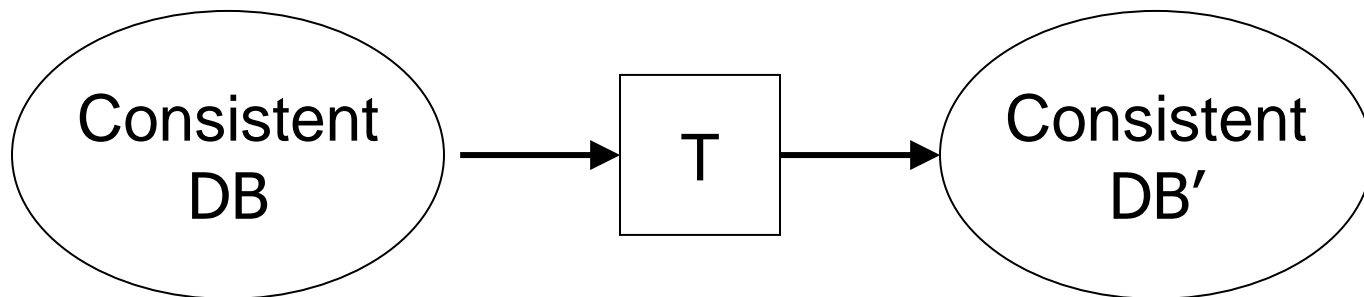
- $\text{TOT} \leftarrow \text{TOT} + 100$



# Solving inconsistencies

## ■ Transaction

- Collection of actions (updating data) that preserve consistency
  - the actions are ordered – it's a sequence.



# Transaction Processing

## ■ Assumption

- If T starts with consistent state and T executes in isolation
- → T leaves DB in a consistent state

## ■ Correctness

- If we finish running transactions, DB is left consistent
- Each transaction sees a consistent DB

# Consistency Violation

## ■ Causes

- Transaction bug

- DBMS bug

- Hardware failure

- E.g., a disk crash during storing updates to accounts

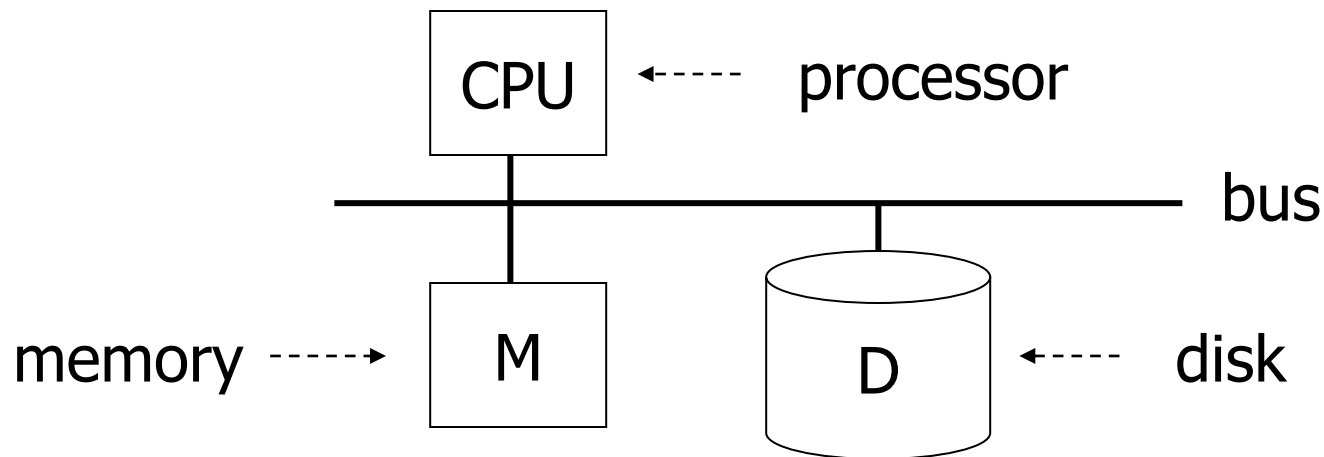
- Data sharing

- E.g.,  
T1: give 10% raise to programmers  
T2: change programmers → systems analysts

# Prevent Consistency Violations

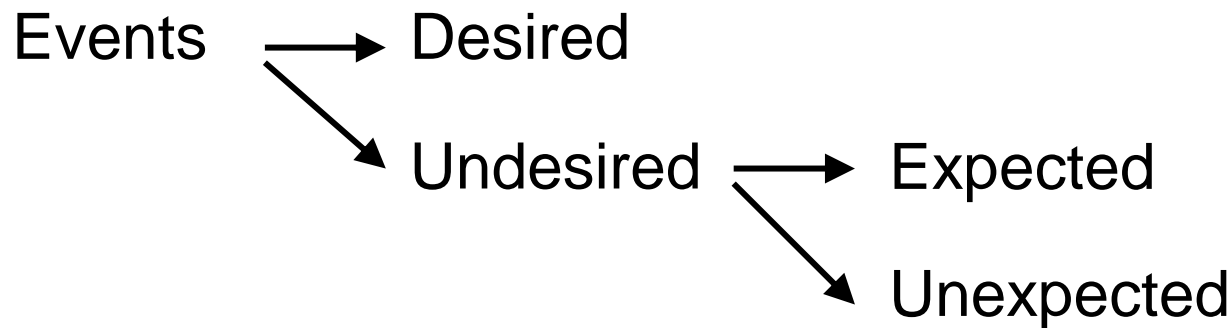
## ■ Failure model

- Identify possible risks
- Handle individual component failures



# Prevent Consistency Violations

- Failure model
  - Categorize risks



# Prevent Consistency Violations

## ■ Events

### □ Desired

- See product manuals... ☺

### □ Undesired expected

- Memory lost
- CPU halts, resets
- Forcible shutdown

### □ Undesired Unexpected (Everything else)

- Disk data is lost
- Memory lost without CPU halt
- Disaster – fire, flooding, ...

# Failure Model

- Approach:

- Add low-level checks
- Redundancy to increase probability model holds

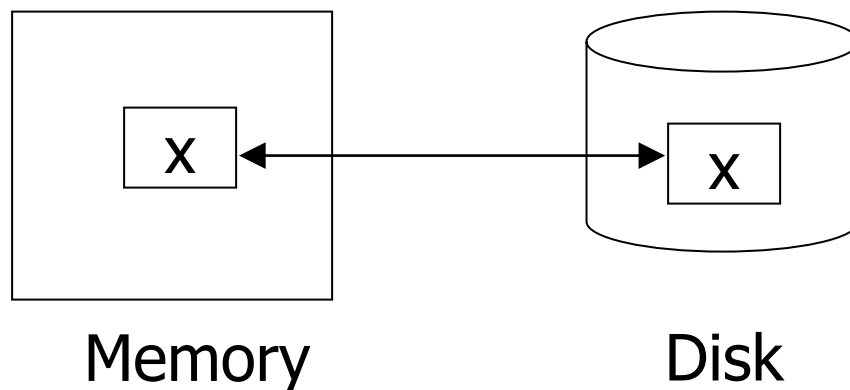
- E.g.,

- Replicate disk storage (stable store, RAID)
- Memory parity, ECC
- CPU checks



# Failure Model

## ■ Focusing on memory



## ■ Key problem

□ Unfinished transactions

□ E.g.,

*Constraint:*

$A=B$

Transaction T1:

$A \leftarrow A \cdot 2$

$B \leftarrow B \cdot 2$

# Transaction

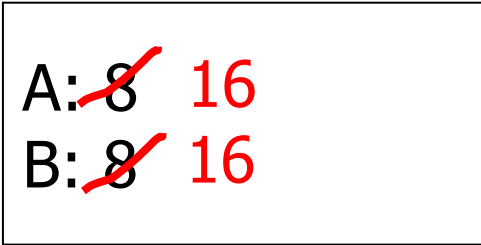
## ■ Elementary operations

- Input (x):      block containing x → memory
  
- Read (x,t):    a. *Input(x)*, if necessary,  
                    b.  $t := \text{value of } x \text{ in block}$
  
- Write (x,t):    a. *Input(x)*, if necessary,  
                    b.  $\text{value of } x \text{ in block} := t$
  
- Output (x):    block containing x → disk

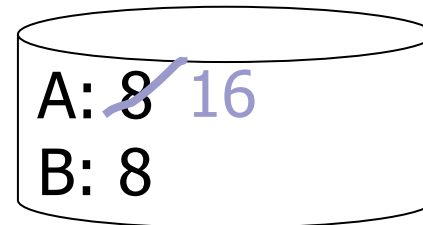
# Example: Transaction T1

```
T1:  Read (A,t);  
      t ← t · 2;  
      Write (A,t);  
      Read (B,t);  
      t ← t · 2;  
      Write (B,t);  
      Output (A);  
      Output (B);
```

Failure!



memory



disk

# Transaction

## ■ Atomicity

- Solution to unfinished transactions
- Execute all actions of a transaction or none at all

## ■ How to implement?

- Log changes done to data
  - i.e., create a journal (file with records about changes)

# Logging

- Transaction produces records of changes into journal
  - Start, End, Output, Write, ...
- Uses:
  - System failure → redo/undo changes following the journal
  - Recovery from backup → redo changes following the journal

# Logging

- During recovery after system failure
  - Some transactions are done again
    - REDO
  - Some transactions are aborted
    - UNDO

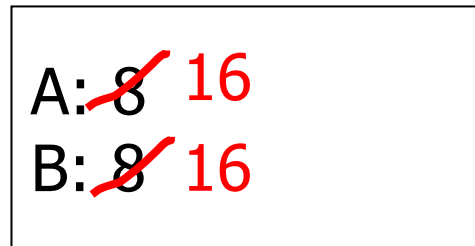
# Undo logging

## ■ Property

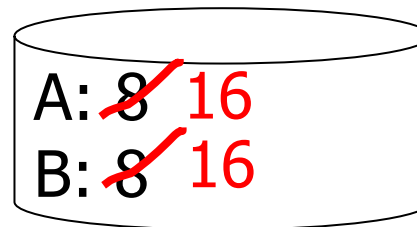
- Changes done in transaction are immediately propagated to disk
  - Original (previous) value is logged.
- ## ■ If not sure (100%) about storing of changes done during finished transaction
- Undo the changes in the data from journal
    - i.e., recover last consistent DB
  - → Transaction has not ever been executed

# Undo logging: Transaction T1

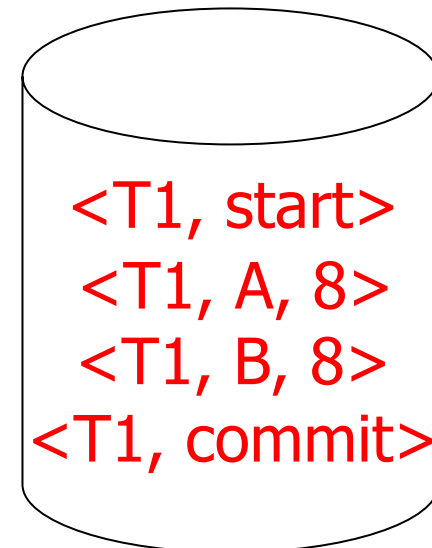
T1: Read (A,t);  
t ← t · 2;  
Write (A,t);  
Read (B,t);  
t ← t · 2;  
Write (B,t);  
Output (A);  
Output (B);



memory



disk



journal

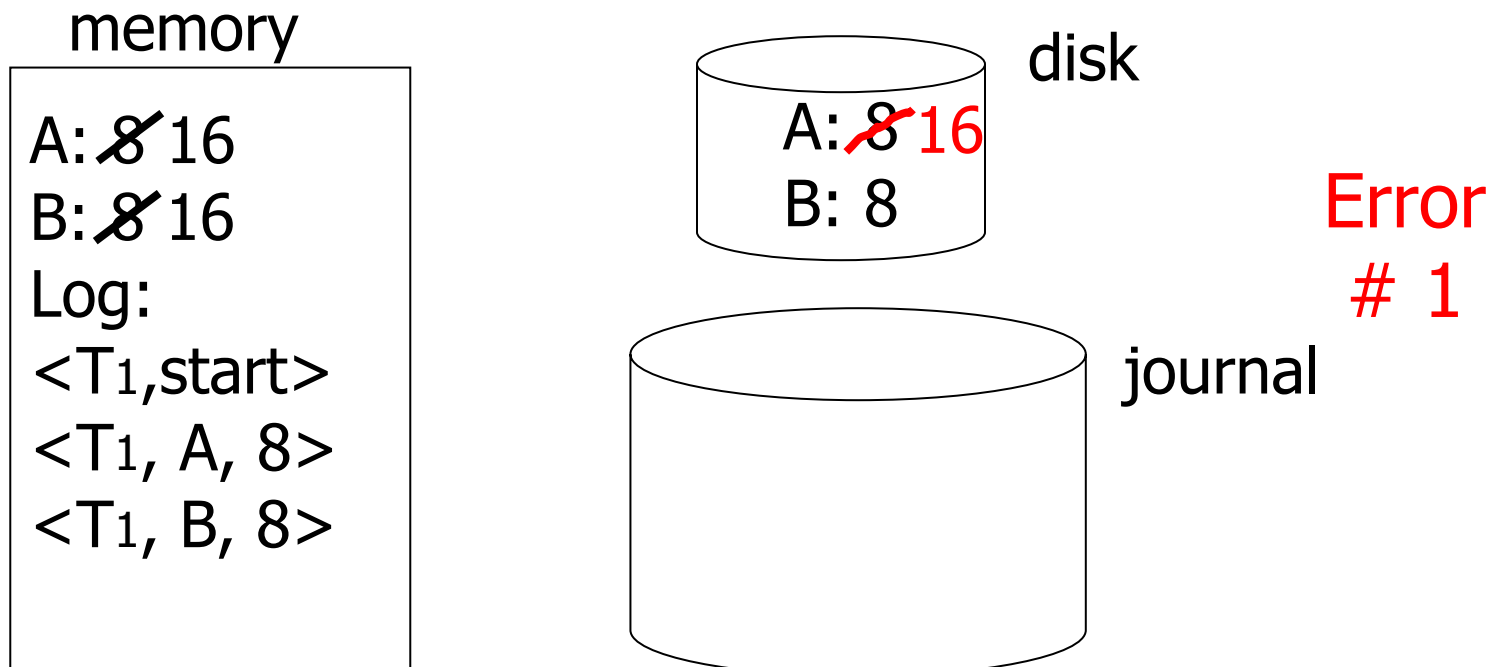
Remark: requiring validity of A=B



# Undo logging

## ■ Inconvenience

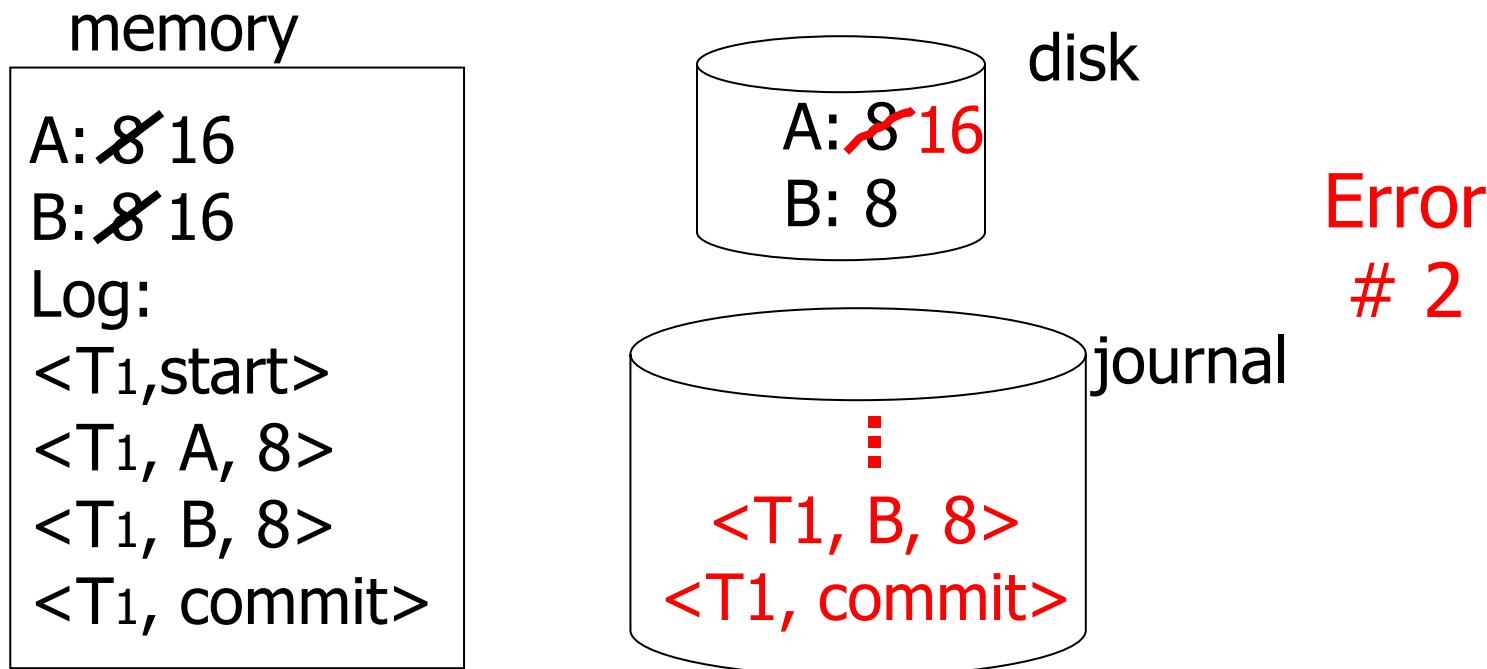
- Logging uses buffer manager too → accumulated in memory, stored to disk later.



# Undo logging

## ■ Inconvenience

- Logging uses buffer manager too → accumulated in memory, stored to disk later.



# Undo logging

## ■ Rules

1. For every action **write**( $X,t$ ), generate undo log record containing old value of  $X$
2. Before  $X$  is modified on disk (**output**( $X$ )), log records pertaining to  $X$  must be on disk
  - i.e., *write-ahead logging* (WAL)
3. Before commit is flushed to log, all writes of transaction must be reflected on disk.

# Undo logging – recovery after failure

- For every  $T_i$  with  $\langle T_i, \text{start} \rangle$  in journal:
  - If  $\langle T_i, \text{commit} \rangle$  or  $\langle T_i, \text{abort} \rangle$  is in log, do nothing
  - Else for every  $\langle T_i, X, v \rangle$  in journal:
    - write( $X, v$ )
    - output( $X$ )
    - write  $\langle T_i, \text{abort} \rangle$  to journal

Is it correct?

# Undo logging – recovery after failure

1.  $S$  = set of transactions
  - with  $\langle T_i, \text{start} \rangle$  in log,
  - but no  $\langle T_i, \text{commit} \rangle$  or  $\langle T_i, \text{abort} \rangle$  in log
2. For each  $\langle T_i, X, v \rangle$  in log
  - in the reverse order do  
(latest  $\rightarrow$  earliest)
  - If  $T_i \in S$ , then write( $X, v$ ) and output ( $X$ )
3. For each  $T_i \in S$ 
  - write  $\langle T_i, \text{abort} \rangle$  to log
    - after successful writing, all output( $X$ ) to disk

# Undo logging – recovery after failure

- Failure during recovery

- No problem

- UNDO can be done repeatedly (is idempotent)
    - Done for unfinished transactions

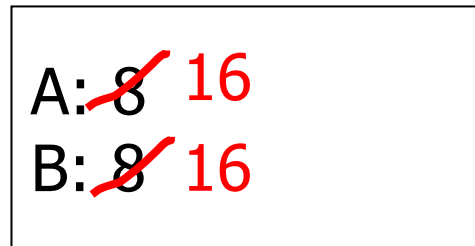
# Redo logging

## ■ Properties

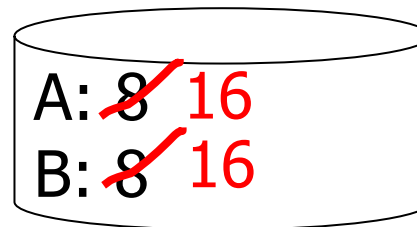
- Logging of new (updated) values
- Changes done by transaction are *stored later*
  - → after transaction's commit
  - i.e., requires storing log records before any change is done to DB.
  - May save some intermediate writes to disk.
- Unfinished transactions are skipped during recovery

# Redo logging: Transaction T1

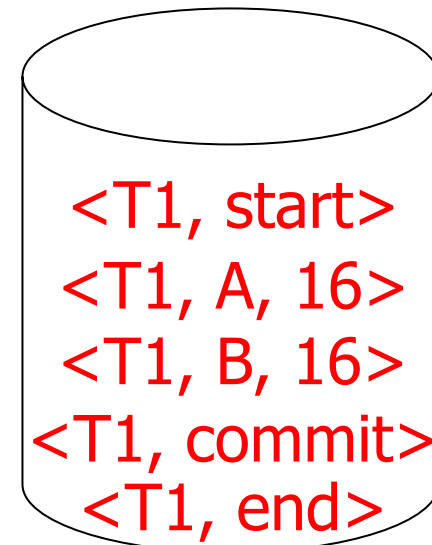
T1: Read (A,t);  
t ← t · 2;  
Write (A,t);  
Read (B,t);  
t ← t · 2;  
Write (B,t);  
Output (A);  
Output (B);



memory



disk



journal



# Redo logging

## ■ Rules

1. For every action **write**( $X,t$ ), generate log record containing a new value of  $X$
2. Before  $X$  is modified on disk (in DB) (**output**( $X$ )), all log records that modified  $X$  (including commit) must be on disk.
3. For transaction modifying  $X$ 
  1. Flush log records to disk
  2. Write updated blocks to disk
  3. Write *end* to journal

# Redo logging – recovery after failure

- For every  $T_i$  with  $\langle T_i, \text{commit} \rangle$  in log, do:
  - For all  $\langle T_i, X, v \rangle$  in log:
    - write( $X, v$ )
    - output( $X$ )

Is it correct?

# Redo logging – recovery after failure

## 1. $S$ = set of transactions

- with  $\langle T_i, \text{commit} \rangle$  in log,
- but no  $\langle T_i, \text{end} \rangle$

## 2. For each $\langle T_i, X, v \rangle$ in log

- Do in forward order  
(earliest  $\rightarrow$  latest)
- If  $T_i \in S$ , then write( $X, v$ ) and output ( $X$ )

## 3. For each $T_i \in S$

- write  $\langle T_i, \text{end} \rangle$  to log

# Combining $\langle T_i, \text{end} \rangle$ Records

- Want to delay DB flushes for hot objects

Say X is branch balance:

T1: ... update X...

T2: ... update X...

T3: ... update X...

T4: ... update X...

Log actions:

write X,  $v_1$

~~output X~~

write X,  $v_2$

~~output X~~

write X,  $v_3$

~~output X~~

write X,  $v_4$

output X

combined  $\langle \text{end} \rangle$  (checkpoint)

# Redo logging – recovery after failure

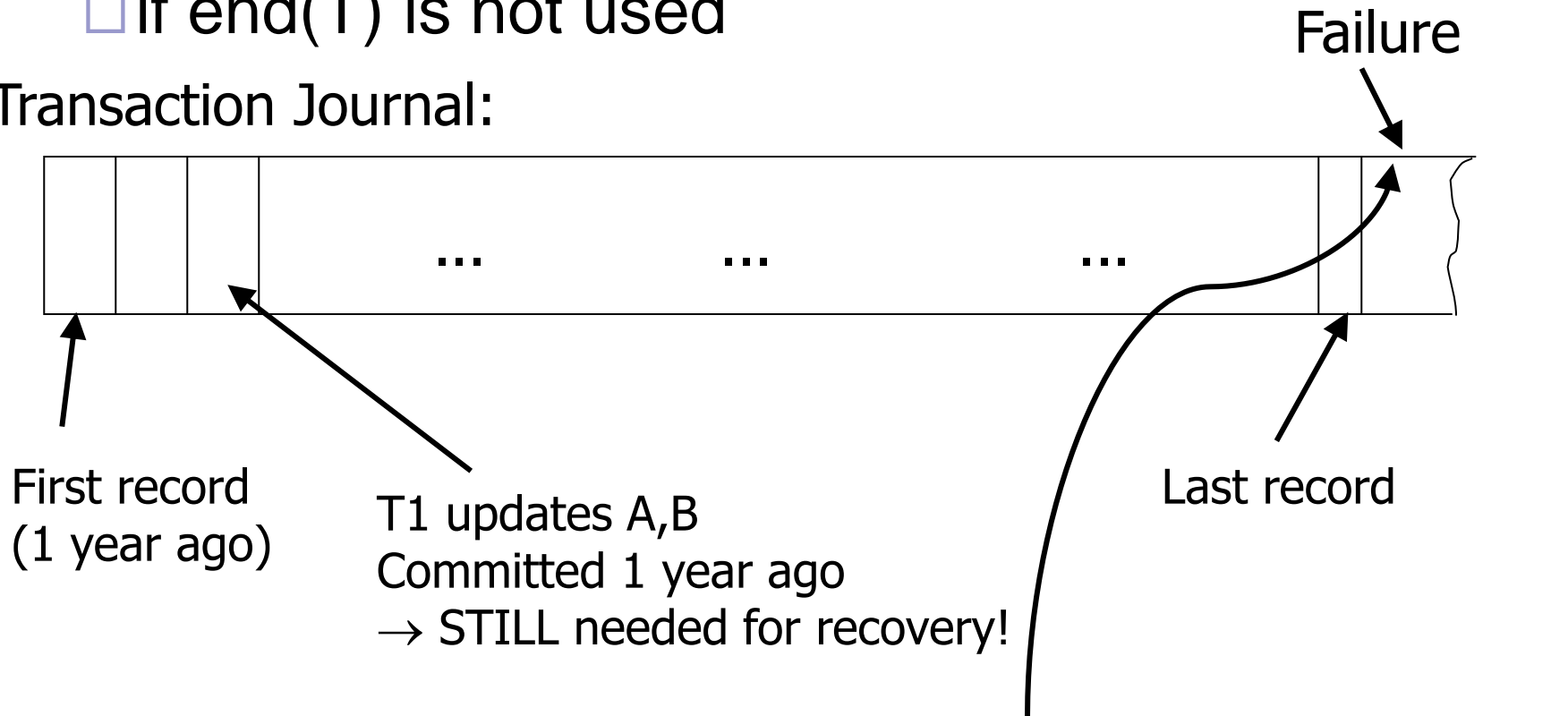
- Storing changes by output( $X$ )
  - If there are more transactions changing  $X$ ,
  - then output( $X$ ) can be done for the last log record  $\langle T_i, X, v \rangle$  only
  - *end* can also be combined for multiple transactions

# Redo logging – recovery after failure

- Recovery is very slow

  - if end(T) is not used

Transaction Journal:



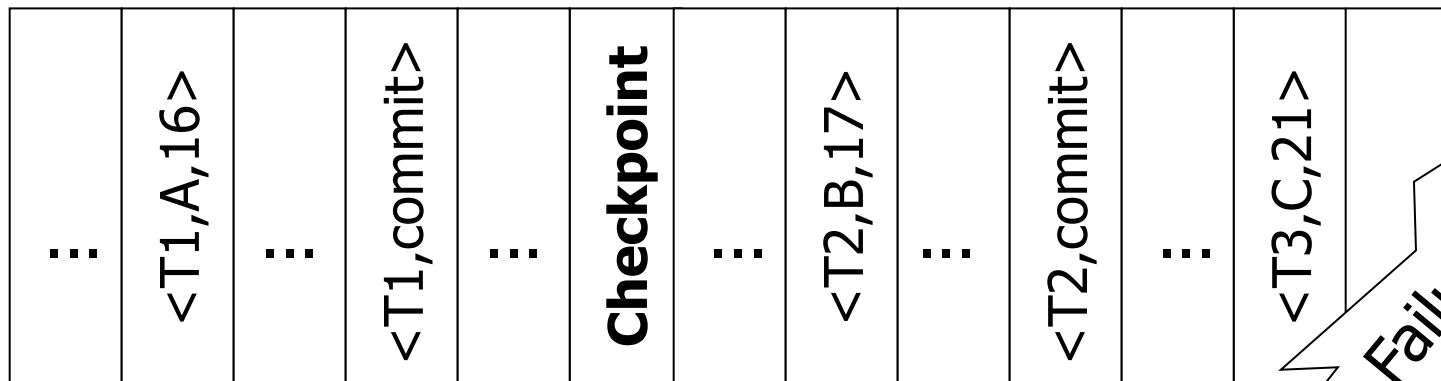
Does DB know what transactions are active here?

# Logging – recovery after failure

- Solution to slowness  
→ checkpoints
- Periodically do:
  1. Do not accept new transactions
  2. Wait until all transactions finish
  3. Flush all log records to disk (log)
  4. Flush all buffers to disk (DB)
  5. Write “checkpoint” record on disk (log)
  6. Resume transaction processing

# Logging – recovery after failure

- Procedure during recovery
  - Locate last checkpoint
  - Start recovery from this place
- Example: redo log



Failure/Crash



# Logging

## ■ Key drawbacks

- Writes to disk are controlled by logging rules and not by accesses to data.
- Undo logging
  - cannot bring backup DB copies up to date
- Redo logging
  - need to keep all modified blocks in memory until commit

## ■ Solution: Undo/Redo logging

- Log record contains old and new value of X:  
<T<sub>i</sub>, x, new X val, old X val>

# Undo/Redo logging

## ■ Rules

- Page  $X$  can be flushed before or after  $T_i$ 's commit
- Log record flushed before corresponding updated page (WAL)
- Flush log records at commit

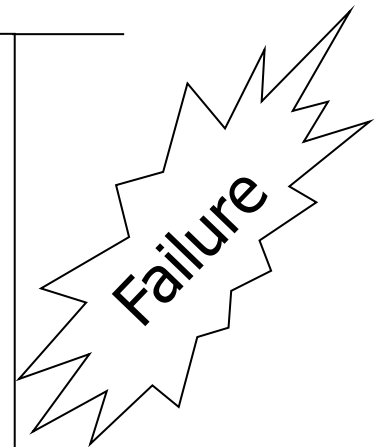
## ■ Recovery

- Finished (committed) transactions are re-done from beginning
- Unfinished transactions are rolled back (un-done) from end

# Undo/Redo logging – recovery

- Example of undo/redo log:

⋮	<checkpoint>	⋮	<T1, A, 11, 10>	⋮	<T1, B, 21, 20>	⋮	<T1, commit>	⋮	<T2, C, 31, 30>	⋮	<T2, D, 41, 40>	
---	--------------	---	-----------------	---	-----------------	---	--------------	---	-----------------	---	-----------------	--



# Checkpoints

## ■ Simple checkpoint

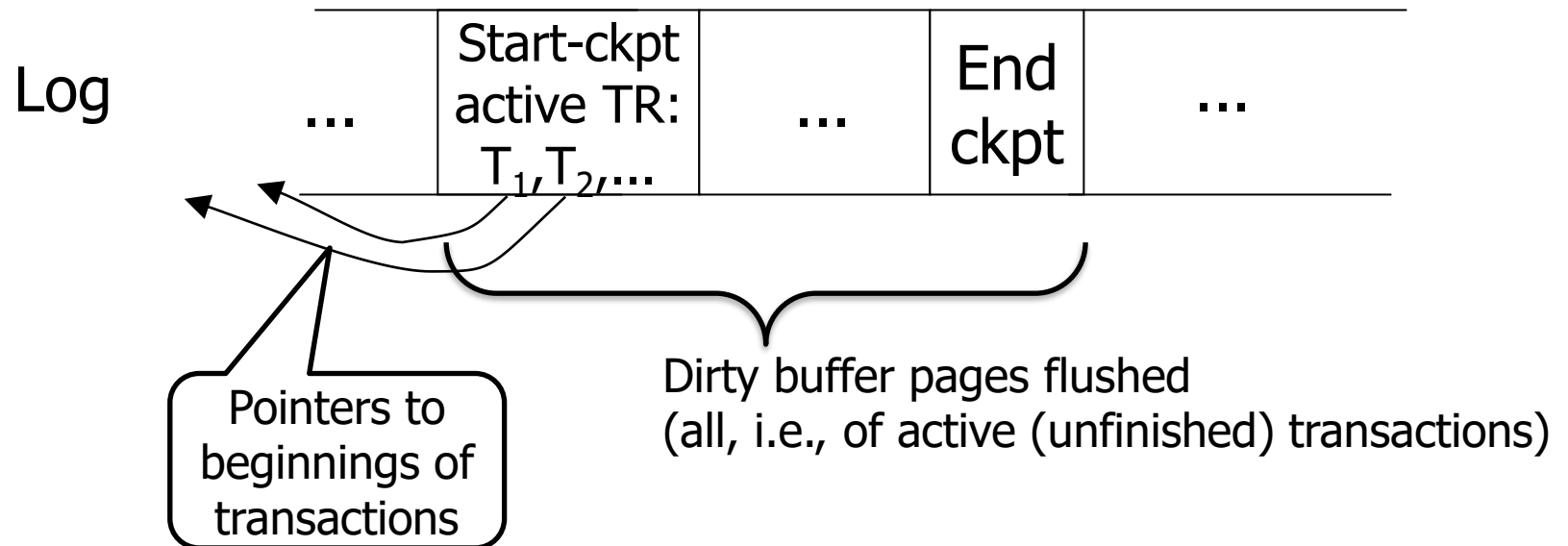
- No transaction can be active during creating checkpoint
- Transaction throughput considerably lowered!

## ■ Solution

- Non-quietescent Checkpoint
  - Register active transactions
  - UNDO/REDO logging:
    - all modified pages (blocks) are flushed to disk

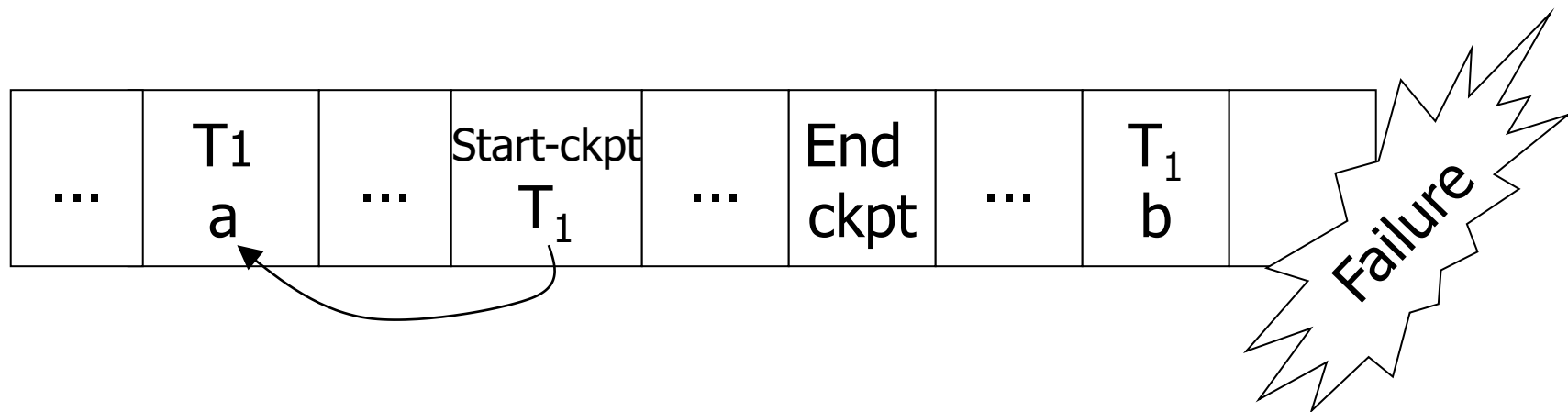
# Non-quiescent Checkpoint

- Store start and end of checkpoint



# Non-quiescent Checkpoint

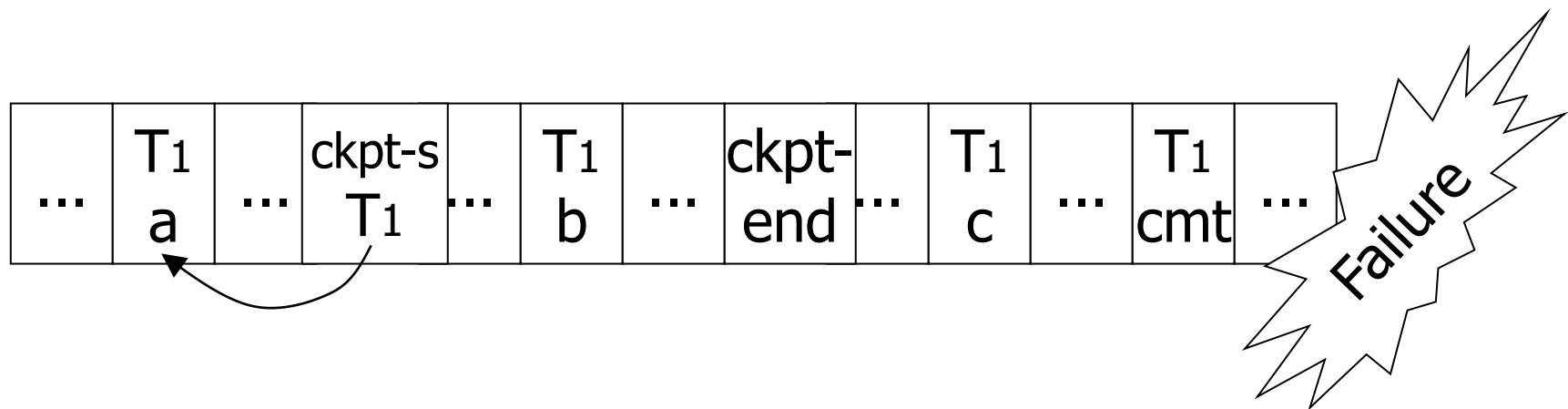
- Recovery 1



- $T_1$  has not been committed  $\rightarrow$  Undo  $T_1$   
(undo changes to  $b$ ,  $a$ )

# Non-quietescent Checkpoint

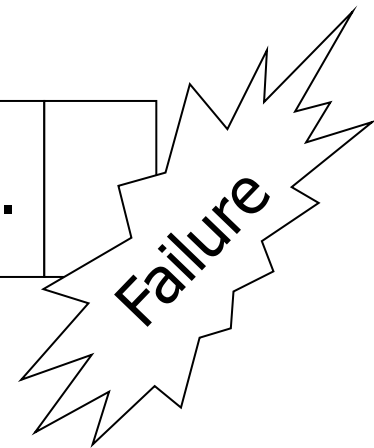
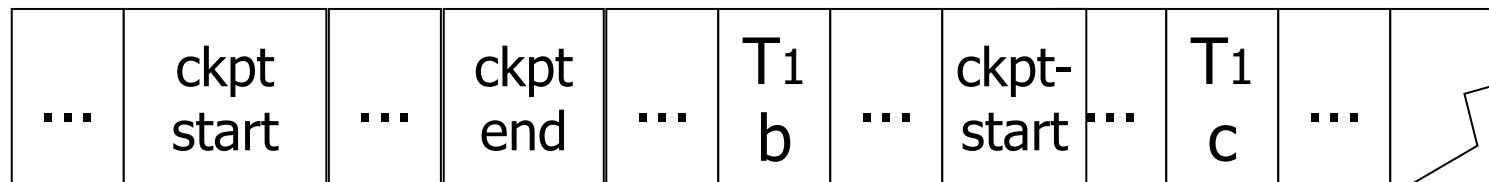
## ■ Recovery 2



- $T_1$  has been committed  $\rightarrow$  Redo  $T_1$  (redo  $b, c$ )

# Non-quietescent Checkpoint

## ■ Recovery 3



- Unfinished checkpoint
  - Locate last *finished* checkpoint
  - Start undo/redo of transactions



# Recovery process

## ■ Backwards pass

(end of log → latest valid checkpoint start)

1. construct set  $S$  of committed transactions
2. undo actions of transactions not in  $S$

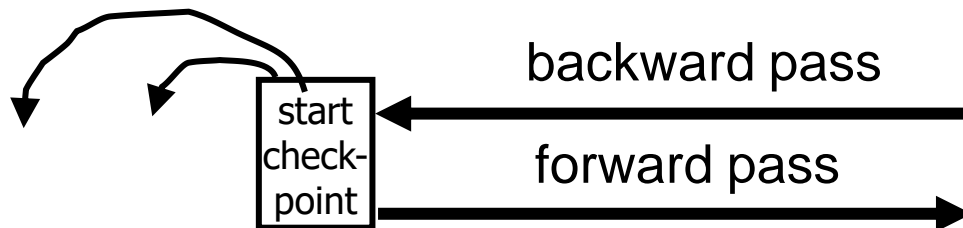
## ■ Remark: Undo pending transactions

- Follow undo chains for transactions in checkpoint active list

## ■ Forward pass

(latest checkpoint start → end of log)

- redo actions of  $S$  transactions (without *end*)



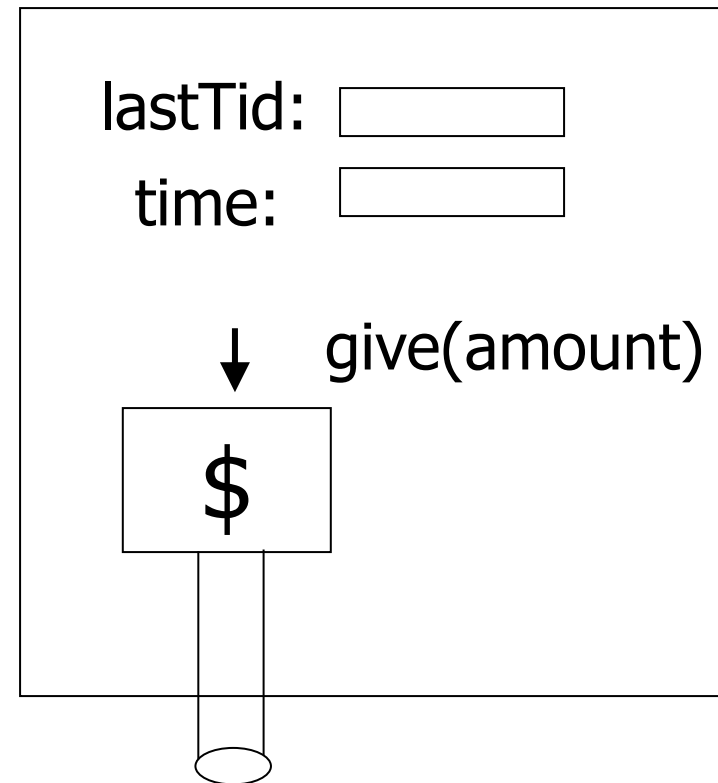
# Real world transaction

- Withdraw cash from ATM
  - Info about bank accounts
  - HW of ATM
- Implementation
  - Transaction in DB
  - Dispense money
- Procedure
  - Do DB transaction, money dispensing after commit.
  - Dispensing should be made idempotent.

# Real world transaction

- After DB transaction, a “signal“ for money dispensing is sent

Give\$\$ (amount, Tid, time)



# Media Failure

- RAID

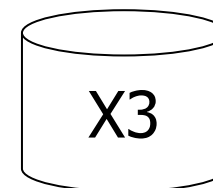
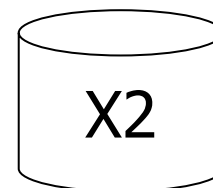
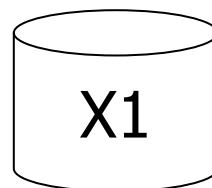
- Make copies of data

- E.g.,

- Keep 3 copies

- Output(X)  
→ three outputs

- Input(X)  
→ three inputs + voting



# Media Failure

- Make copies of data

- Other solution

- Keep 3 copies

- Output(X)

- three outputs

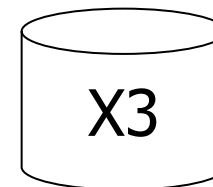
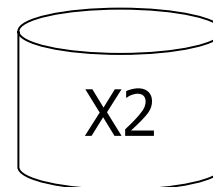
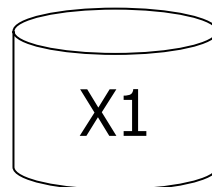
- Input(X)

- read from first (if ok, continue)

- read from second, ...

- Assumption

- bad data can be detected



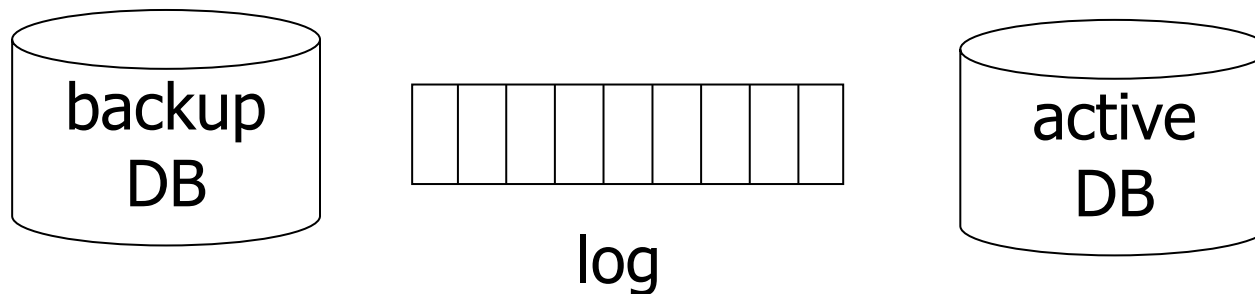
# Media Failure

- DB backup (dump)

- Recover DB backup

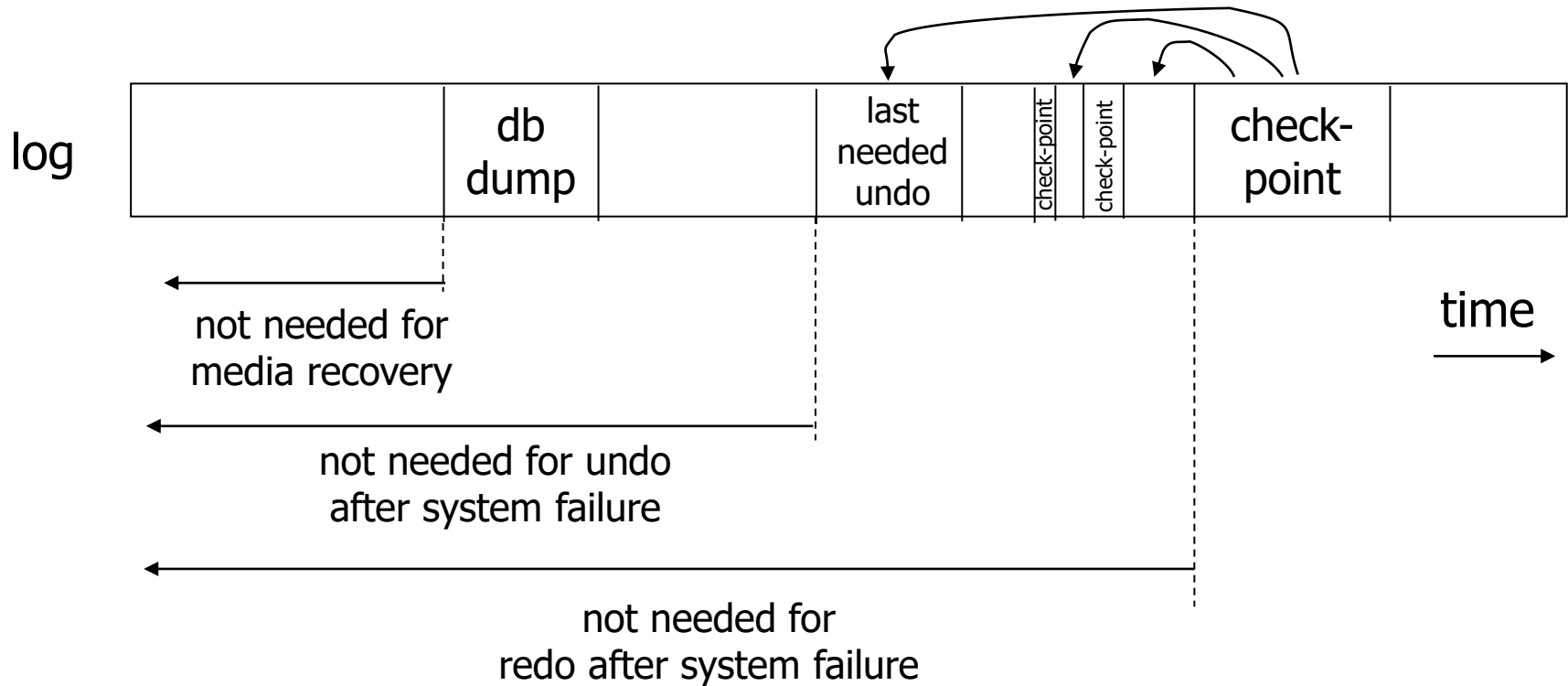
- Apply log

- Use redo entries of each transaction not finished at the backup time



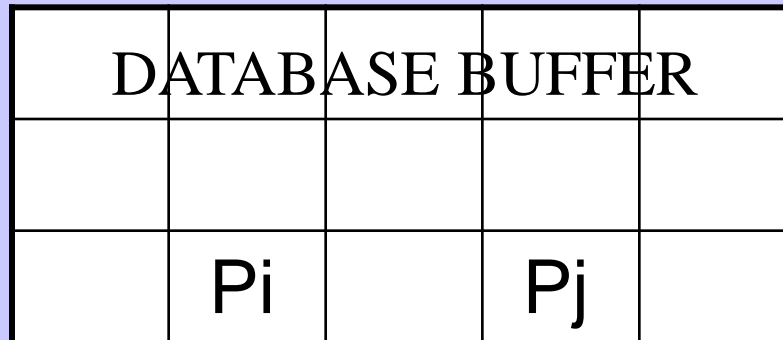
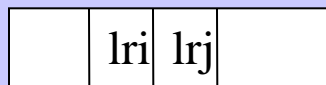
# Discarding Log

- When can log be discarded?
  - In case of UNDO/REDO logging



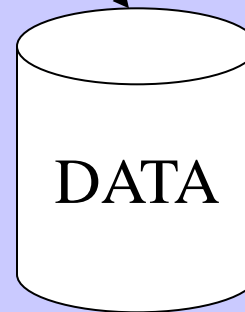
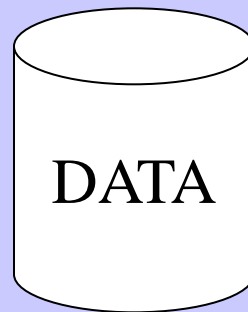
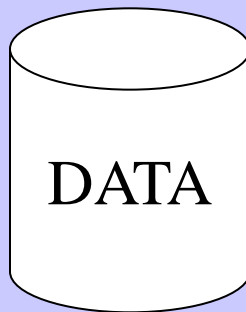
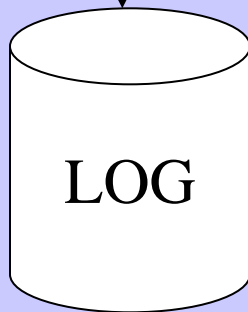
## UNSTABLE STORAGE

LOG BUFFER



WRITE  
log records before commit

WRITE  
modified pages after commit



RECOVERY

STABLE STORAGE

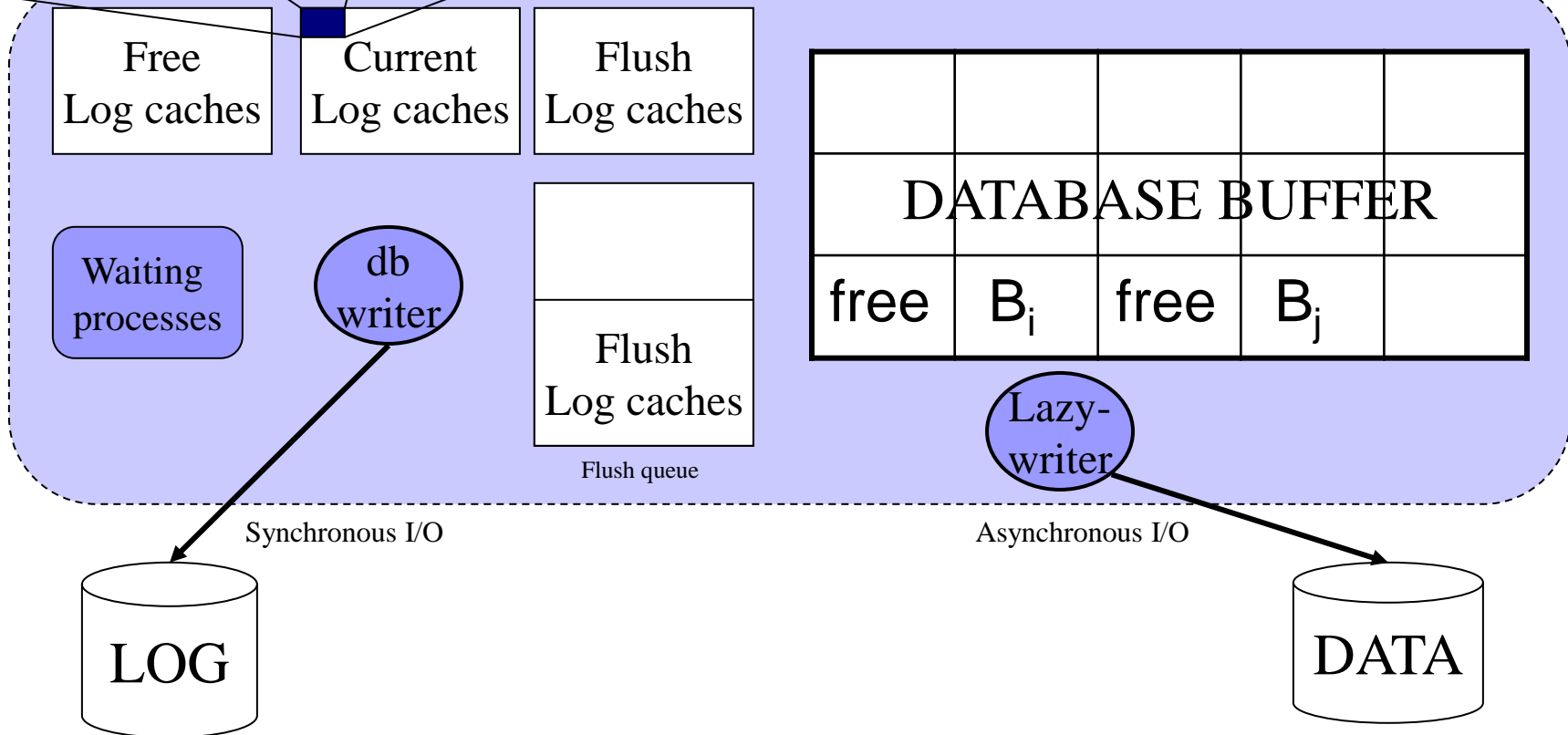


# Logging in SQLServer 2000

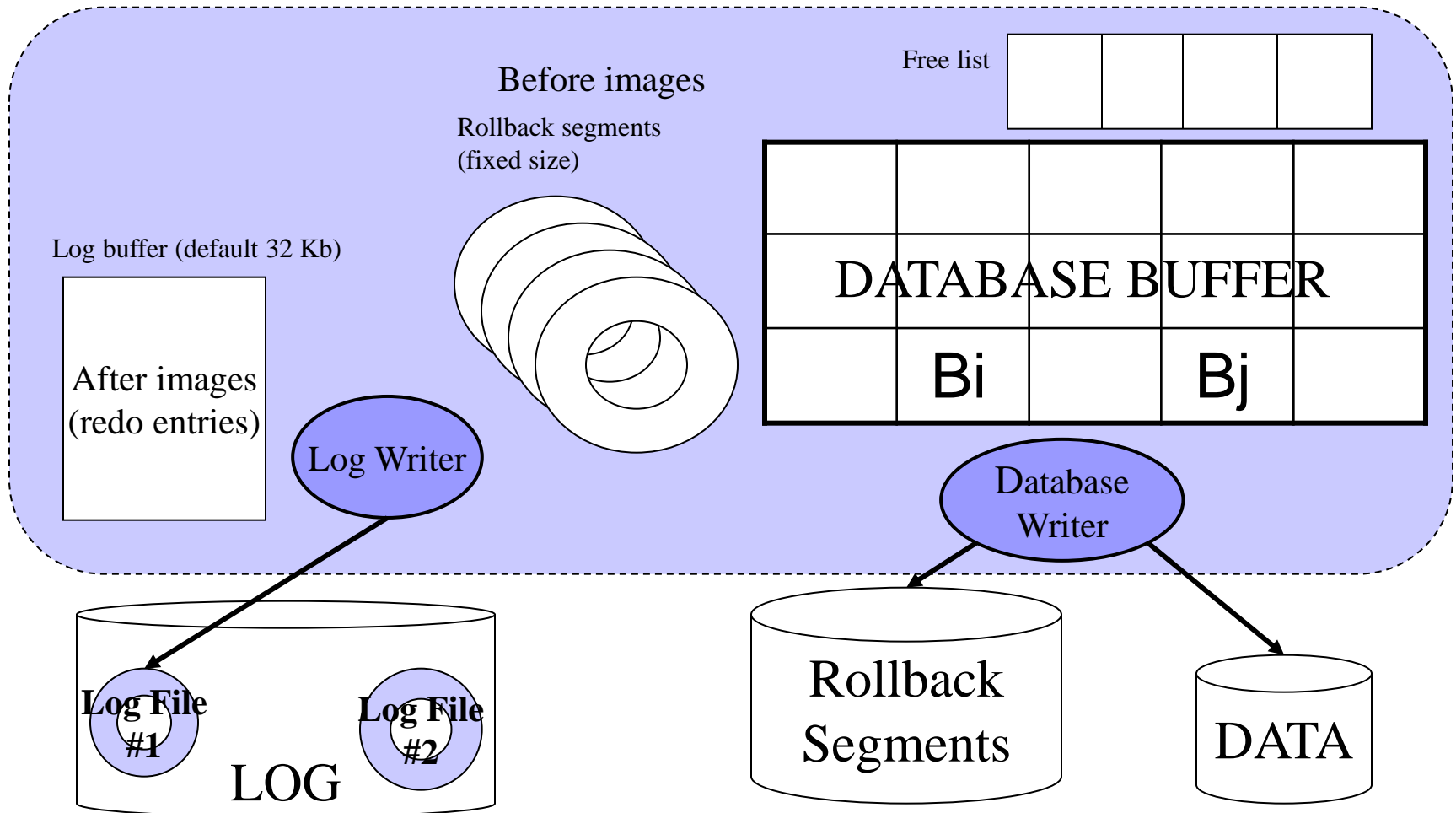
DB2 v7 uses similar schema

Log entries:

- LSN
- before and after images or logical log



# Logging in Oracle 8i



# Storing Log

- On dedicated disk
- Log records are stored sequentially
- Sequential writes are much faster than random ones (on a magnetic disk)

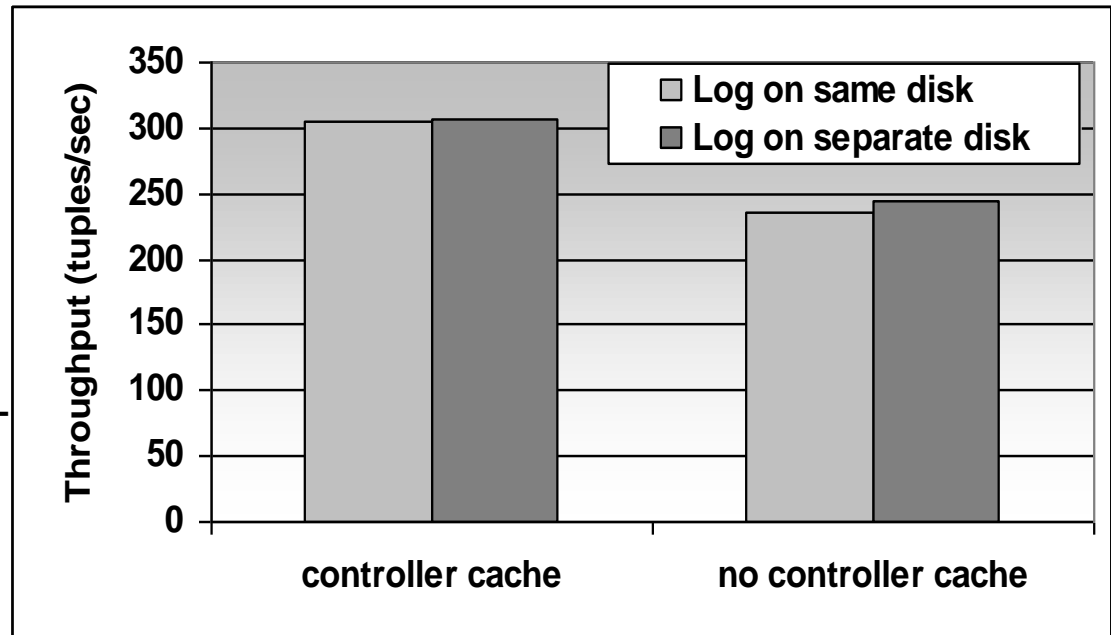
Disk for logging should not store any other data  
+ sequential I/O  
+ loss of log is not dependent on loss of DB

# Storing Log

300 000 transactions  
Each transaction = 1x INSERT

## DB2 v7.1 server

5% improvement when log  
stored on dedicated disk



Controller Cache diminishes negative impact of non-dedicated disk  
HW: middle server, Adaptec RAID controller (80Mb RAM), 2x18Gb disk.

# Flushing Buffers

## ■ Flushing dirty page

- When a threshold of modified pages is reached (Oracle 8)
- When the ratio of free pages drops below a threshold (less than 3% in SQLServer 7)
- After checkpoint
- Periodically

# Creating Checkpoints

Performance influence (decreased throughput)

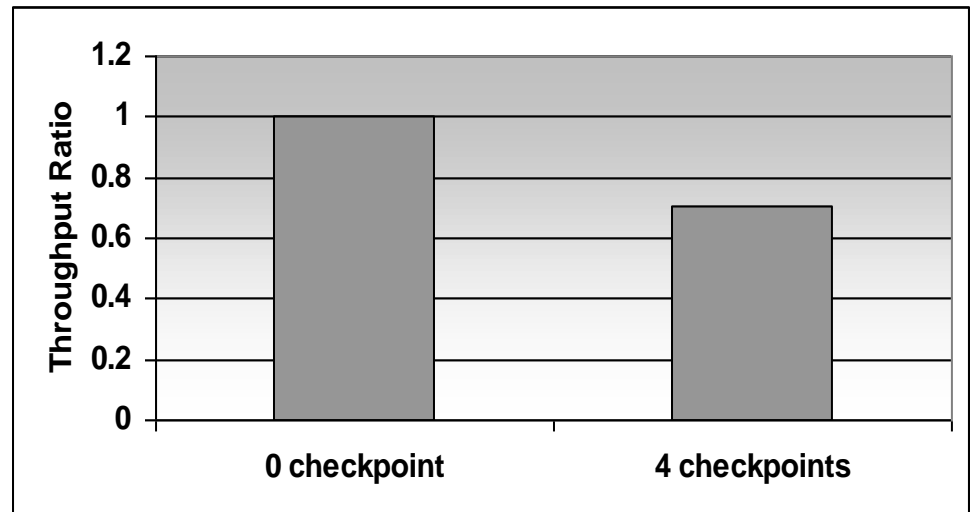
Reduces size of log

Shortens time to recover after failure

300 000 transactions

Each transaction = one INSERT command

Oracle 8i, Windows 2000



# Lecture Takeaways

## ■ Data consistency

- One source of problems: failures
  - Solutions: (i) logging; (ii) redundancy
- Another source of problems: data sharing
  - Solution: (i) Locking data during transactions
    - Not done in this course...

## ■ Logging

- Know principles and limitations
- Understand checkpoints
- Be able to do recovery