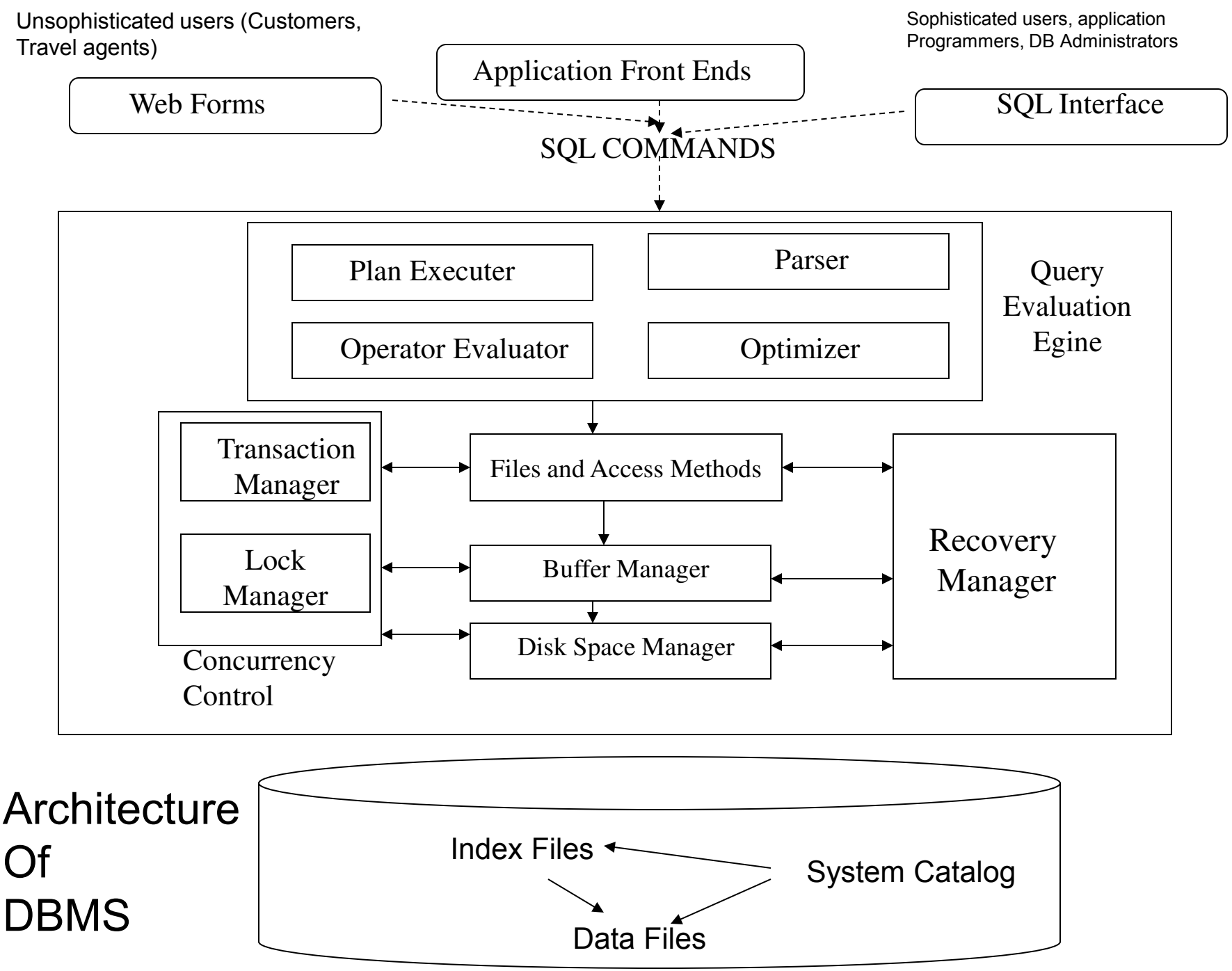
Crash Recovery

(Chapter 8 of Database System Implementation by Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom)



Architecture Of DBMS

Background

- Controlling the access to data
  - Data must be protected in the face of a system failure (Recovery Manager)
  - Data must not be corrupted due to data modifications (Concurrency control)

Outline

- **Issues and Models for Resilient Operation**
- Undo logging
- Redo logging
- Undo/Redo logging
- Protecting against media failures

Issues and Models for Resilient Operations

- We focus on system failures

Failure Modes

- Erroneous data entry
  - Mistyping
    - Put proper integrity constraints
- Media failures
  - Failures of disk
    - Use one of the RAID (Redundant Array of Inexpensive Disks) scheme
    - Maintain an archive with a tape or disk
    - Keep redundant copies of the database online
- Catastrophic failures
  - Explosions or fires
    - Keep redundant and distributed copies.

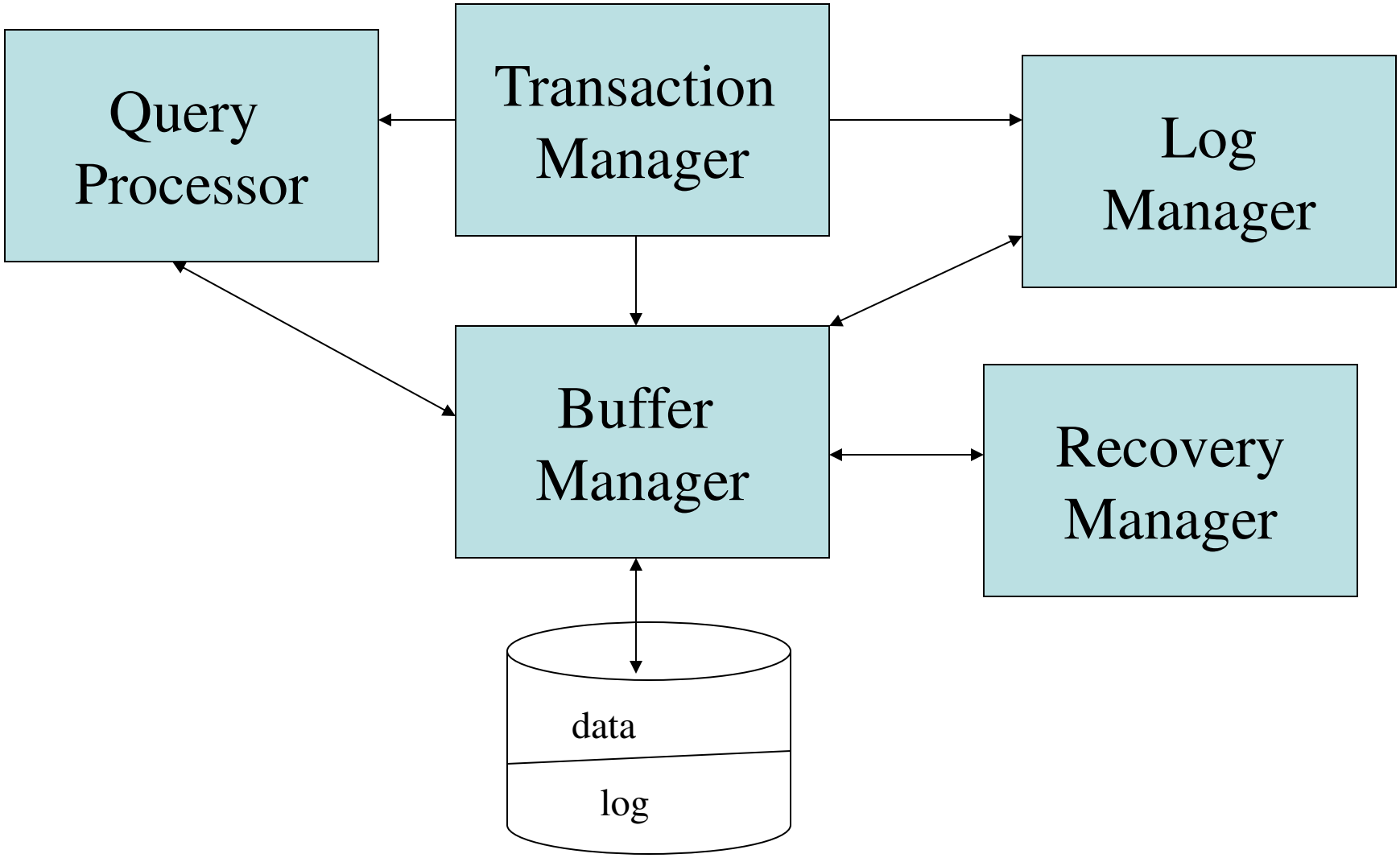
Failure modes: System failures

- Transactions
  - Query and modify the database
  - Sequence of modifications
  - State of a transaction
    - Current place the transaction is being executed and the values of local variables.
- System failures
  - Power outages, main memory is volatile
  - Software error may erase the main memory
- When main memory is lost the state of the transaction is lost.
  - It is difficult to tell about the modifications
  - Rerunning the transaction may not help
  - Logging is the solution.

More about transactions

- Transaction is a unit of database operations.
- SQL
  - Transaction starts as soon as operations are executed
  - End with an explicit COMMIT or ABORT
- Transaction manager assures that transactions are executed correctly.
  - Issues signals to log manager
    - To store the information in the log.
  - Assuring that concurrently executing transactions do not interfere with each other in ways that introduce errors. (next chapter)

Log manager and Transaction manager



The log manager and transaction manager

- Transaction manager
  - sends messages about the actions of transactions to the log manager, to the buffer manager when it is possible to copy the data to disk.
  - To the query processor to execute the queries that comprise a transaction.
- Log manager maintains the log
  - It deals with the buffer manager
- Recovery manager
  - When there is a crash recovery manager is activated.
  - It examines the log and uses it to repair the data
  - Always access to disk is through buffer manager.

Correct Execution of Transactions

- Database is composed of elements
- Element
  - Relations/class
  - Disk blocks or pages
  - Individual tuples of relation or objects.
- We consider the elements as tuples.
  - Disk blocks/pages are normally considered.
  - Matches with the size of data transfer.
- Consistent state is the value of each element.
  - Satisfies all integrity constraints.

Assumptions about transactions

- Correctness principle.
  - If executed lone, it starts in a consistent state and leaves the database in a consistent state, when the transaction ends.
- Transaction is atomic
  - It must be executed as a whole or not at all.
- Transactions execute concurrently may lead to inconsistent state, so control is needed. (Concurrency control)

Review: The ACID properties

- Atomicity: All actions in the Xact happen, or none happen.
- Consistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- Isolation: Execution of one Xact is isolated from that of other Xacts.
- Durability: If a Xact commits, its effects persist.

- Question: which ones does the Recovery Manager help with?

Atomicity & Durability (and also used for Consistency-related rollbacks)

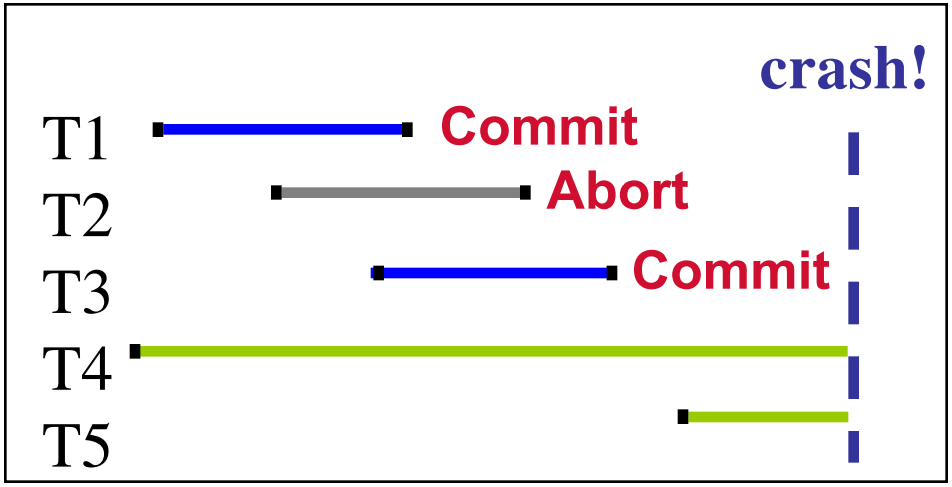
Motivation

- Atomicity:
  - Transactions may abort ("Rollback").
- Durability:
  - What if DBMS stops running? (Causes?)

v Desired state after system restarts:

– T1 & T3 should be durable.

– T2, T4 & T5 should be aborted (effects not seen).



Primitive Operations of Transactions

- Transaction interact with the database
  - The space of disk blocks holding the database elements
  - Virtual or main memory address space managed by buffer manager
  - The local address space of a transaction.
- Reading
  - Data is brought to main memory buffers, the buffer(s) can be read by transaction into its own address space.
- Writing
  - New value is created its own space.
  - The value is copied to buffers

Primitive Operations of a Transaction

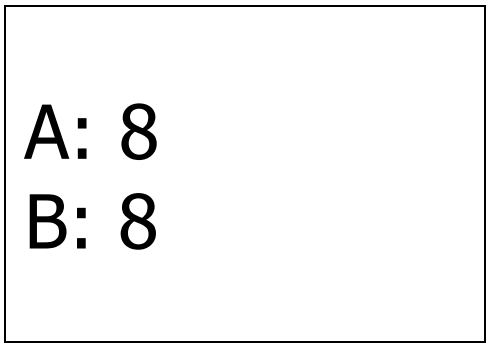
- The buffers may not be copied immediately.
  - The decision is the responsibility of buffer manager
  - We must force the buffer manager to disk at appropriate time.
  - May result into more disk I/Os!
- To reduce the disk I/Os, database systems allow a change to exist only in a volatile main memory storage for a certain periods of time.

Primitives..

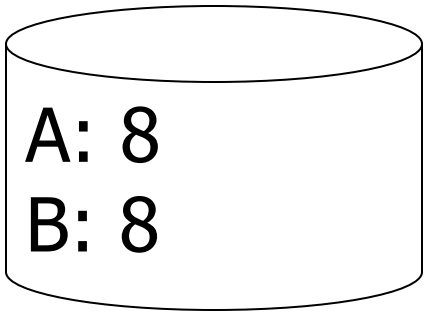
- INPUT(X): Copy the disk block containing disk block X
- READ(X,t): Copy the database element X to the transaction's local variable t.
  - Read(X,t) calls INPUT(X).
- WRITE(X, t): Copy the value of local variable t to database element X in a memory buffer.
  - WRITE(X,t) calls INPUT(X), if X is not in memory buffer.
- OUTPUT(X): Copy the buffer containing to disk.
- INPUT and OUTPUT operations are issued by buffer manager.
- READ/WRITE are issued by transaction.
- Assumption
  - A database element is no larger than a single block.

Example

T_1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



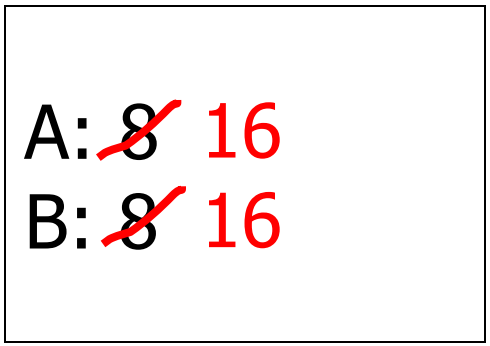
memory



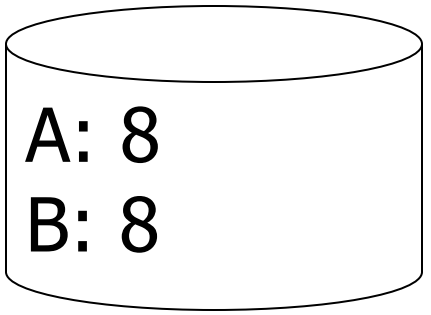
disk

Example

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



memory

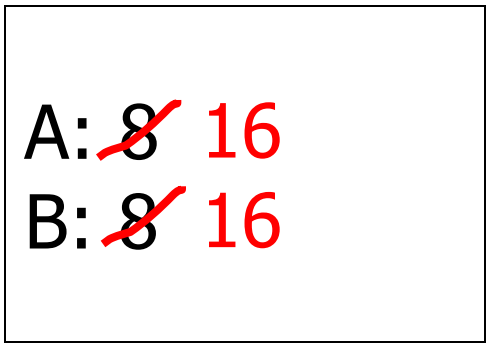


disk

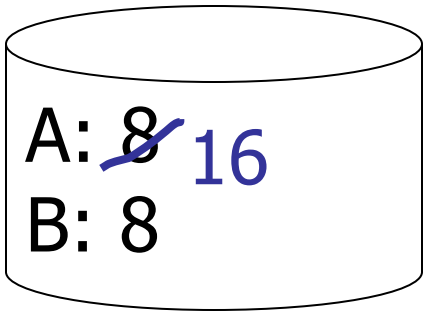
Example

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

Atomicity

- Need atomicity: execute all actions of a transaction or none at all

One solution: undo logging (immediate modification)

Outline

- Issues and Models for Resilient Operation
- **Undo logging (Immediate Modification)**
- Redo logging (Delayed modification)
- Undo/Redo logging (Combination)
- Protecting against media failures

Logging

- A log is a sequence of log records
  - Each tells what a transaction has done.
- When a system is crashed, log is consulted.
  - Committed: new values are written into database
  - Uncommitted: old values are restored into the database.

Log records

- Log is a file opened for appending only
- Log manager records each important event.
- Log records are written into main memory soon as is feasible.
- <START T>: Transaction T has begun
- <COMMIT T>: T has completed successfully.
  - Any changes to the database should appear on disk.
  - But, it may not be. But, log manager should force.
- <ABORT T>: T could not complete successfully.
  - TM must ensure that such changes never appear on disk. Effect on disk has to be cancelled if they do.

Undo Logging

- Uncommitted transaction effects are undone.

Undo logging rules

- U1: If T modifies X, then write <T, X,v> where v is original value of X to disk before the new value of X written to disk.

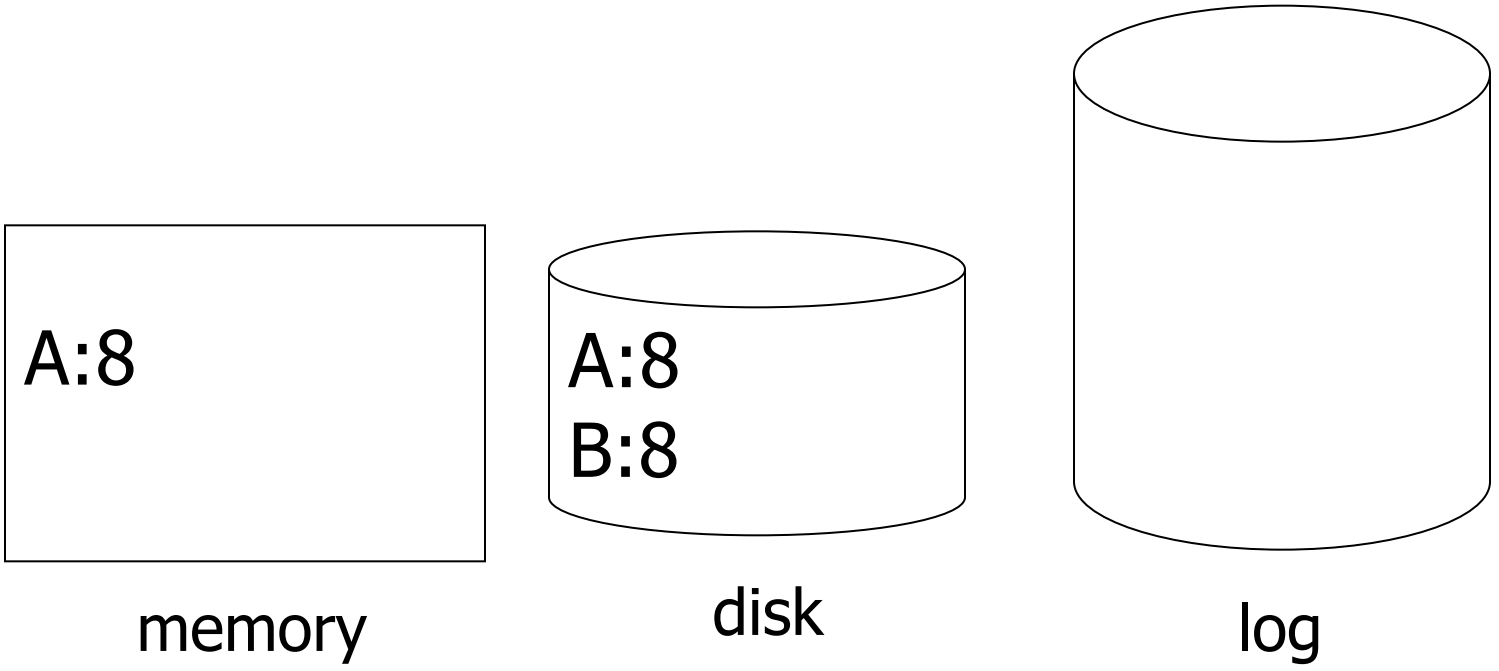
  (write ahead logging: WAL)
- U2: Before commit is flushed to log, all writes of transaction must be reflected on disk.
- Summary
  - Write the log records
  - Write the data elements
  - Write the COMMIT log record.

FLUSH Log Command

- To force log records to disk, log manager issues a flush-log command.
  - Tells the buffer manager to copy to disk any log blocks that have not previously been copied to disk or that have been changed since they were last copied.

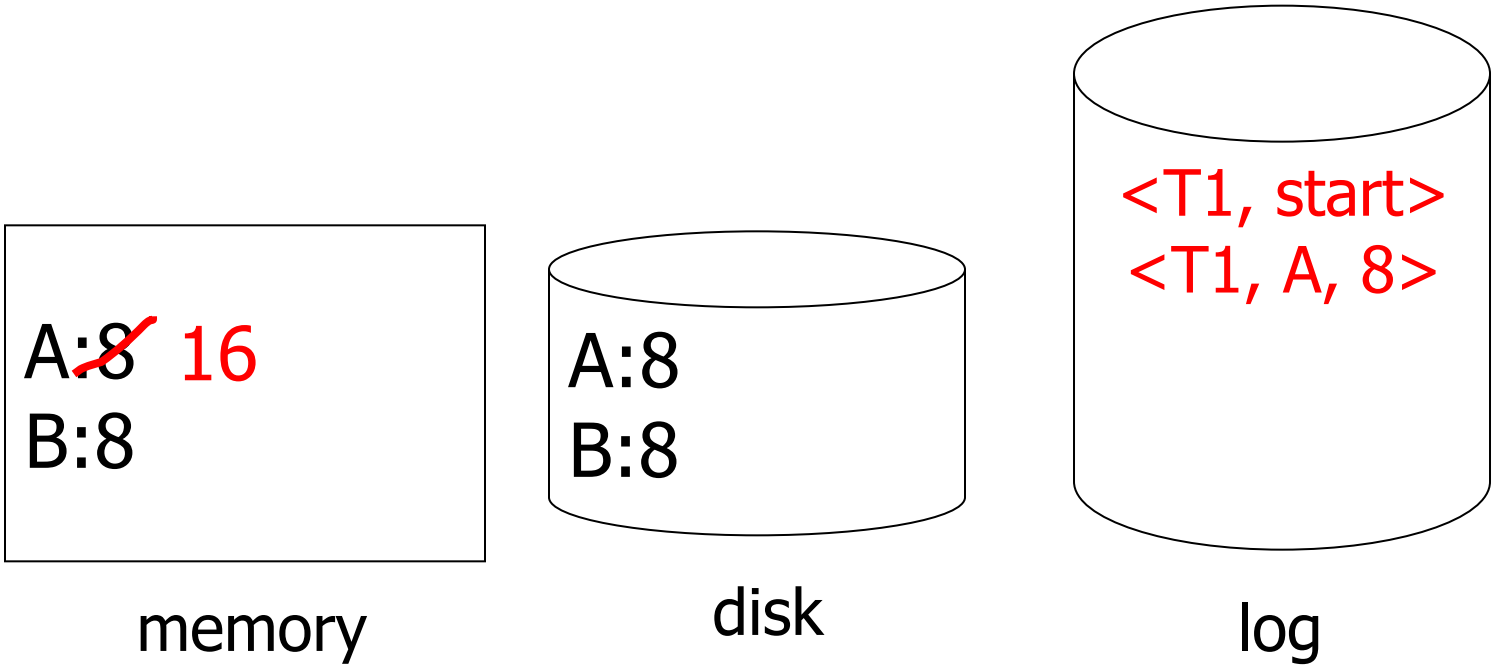
Undo logging (Immediate modification)

T1: Read (A,t); A=B



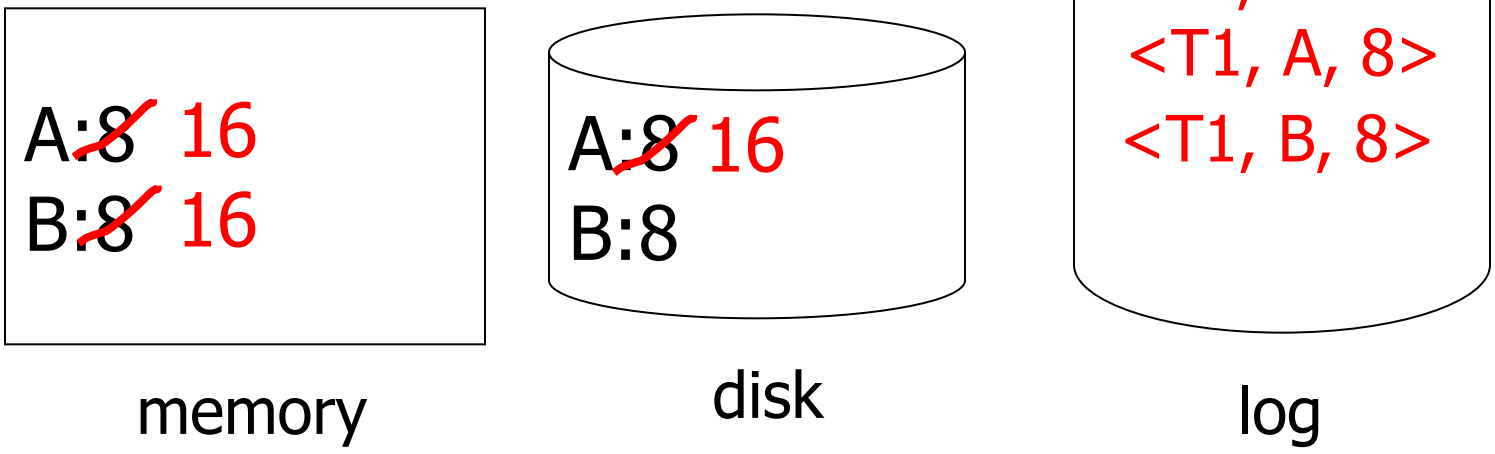
Undo logging (Immediate modification)

T1: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2



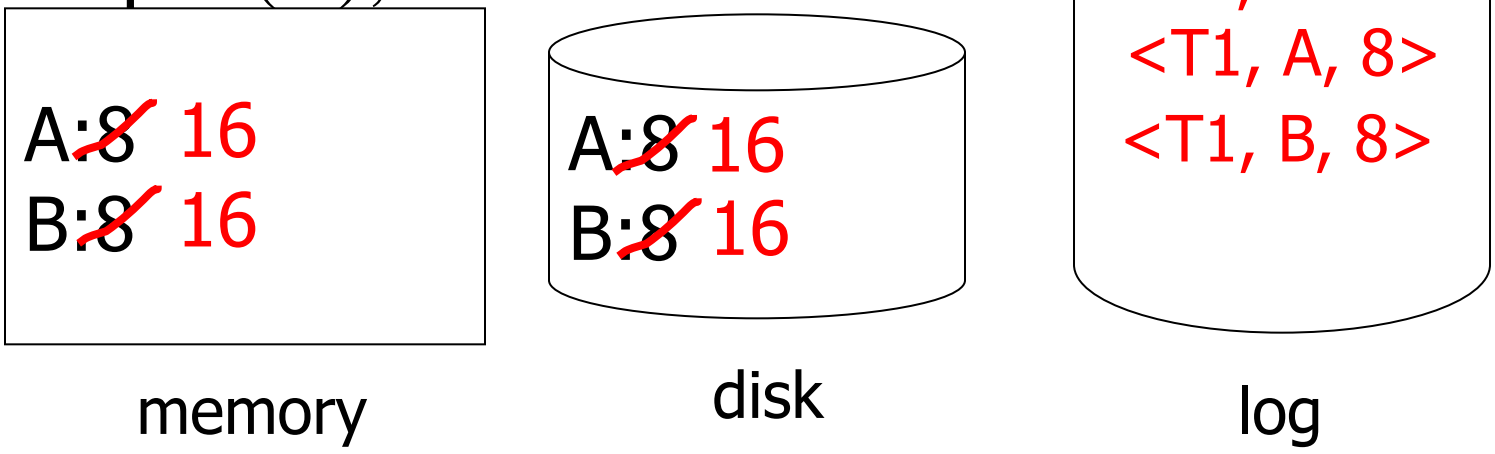
Undo logging (Immediate modification)

T1: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Flush Log
Output (A);



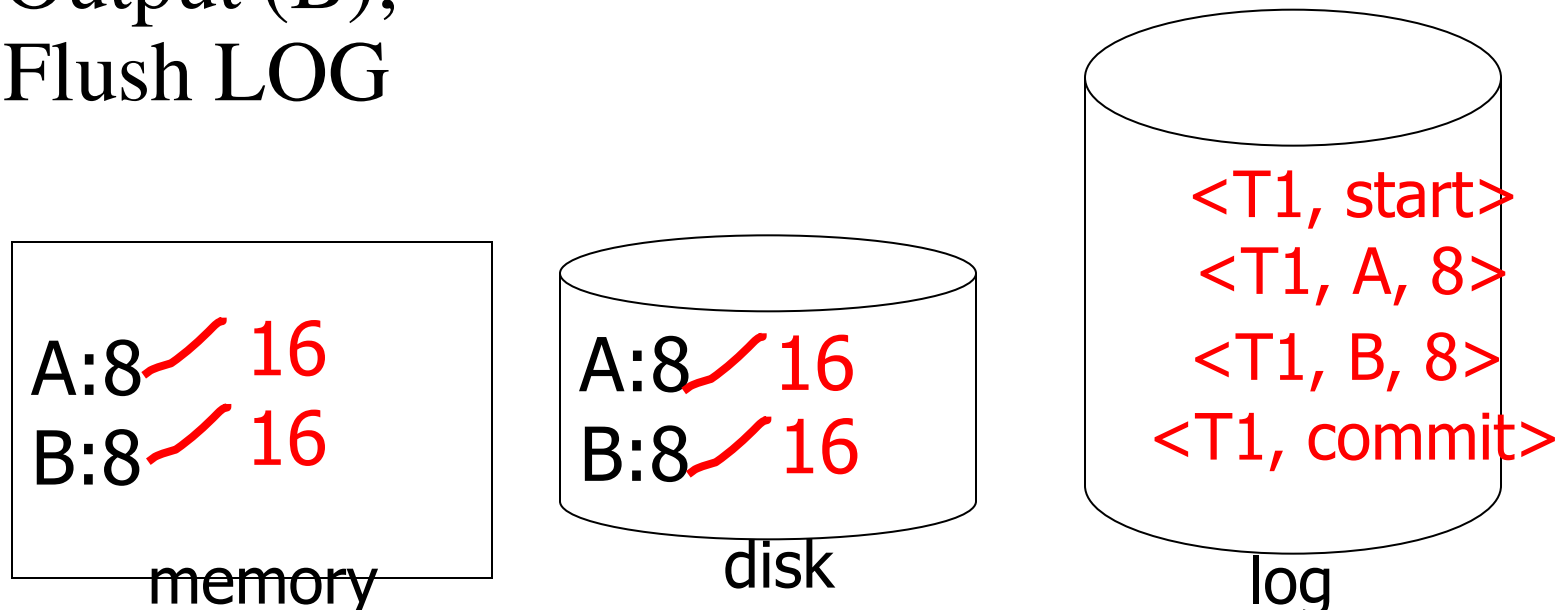
Undo logging (Immediate modification)

T1: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Flush Log
Output (A);
Output (B);



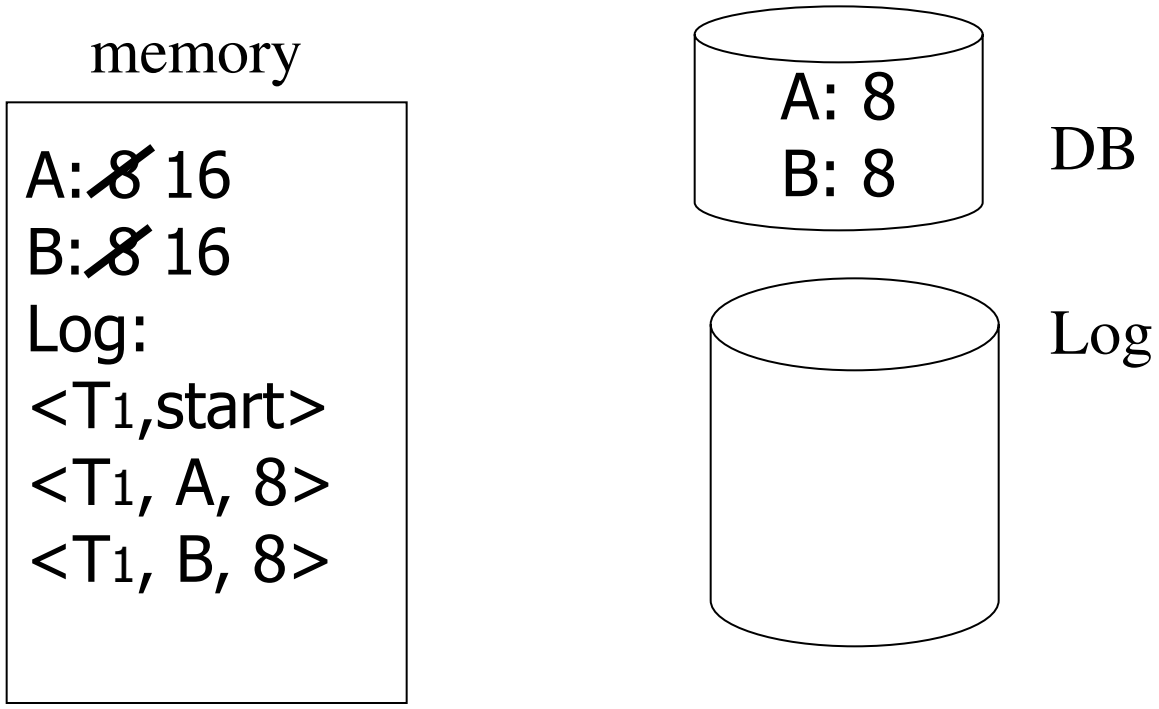
Undo logging (Immediate modification)

T1: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Flush Log
Output (A);
Output (B);
Flush LOG



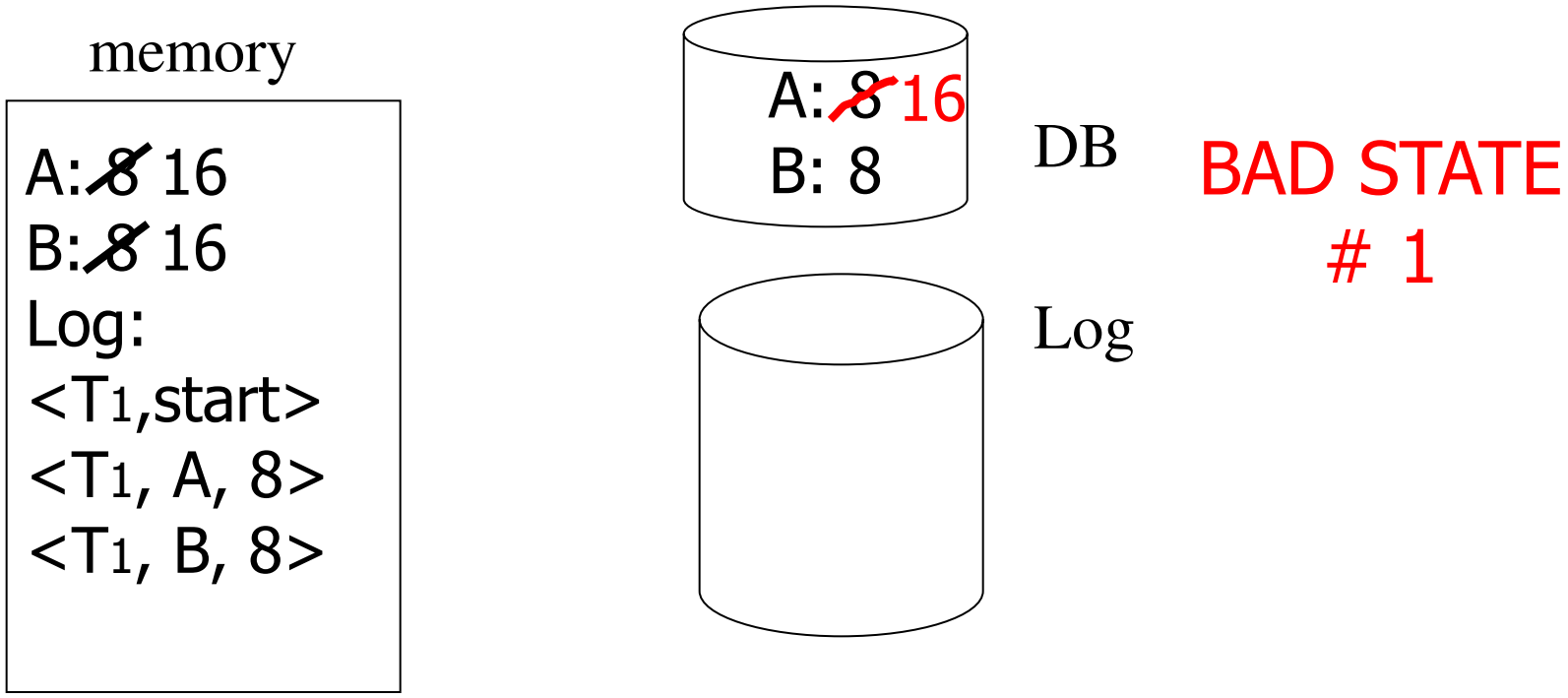
Flushing the Log

- Log is first written in memory
- Not written to disk on every action



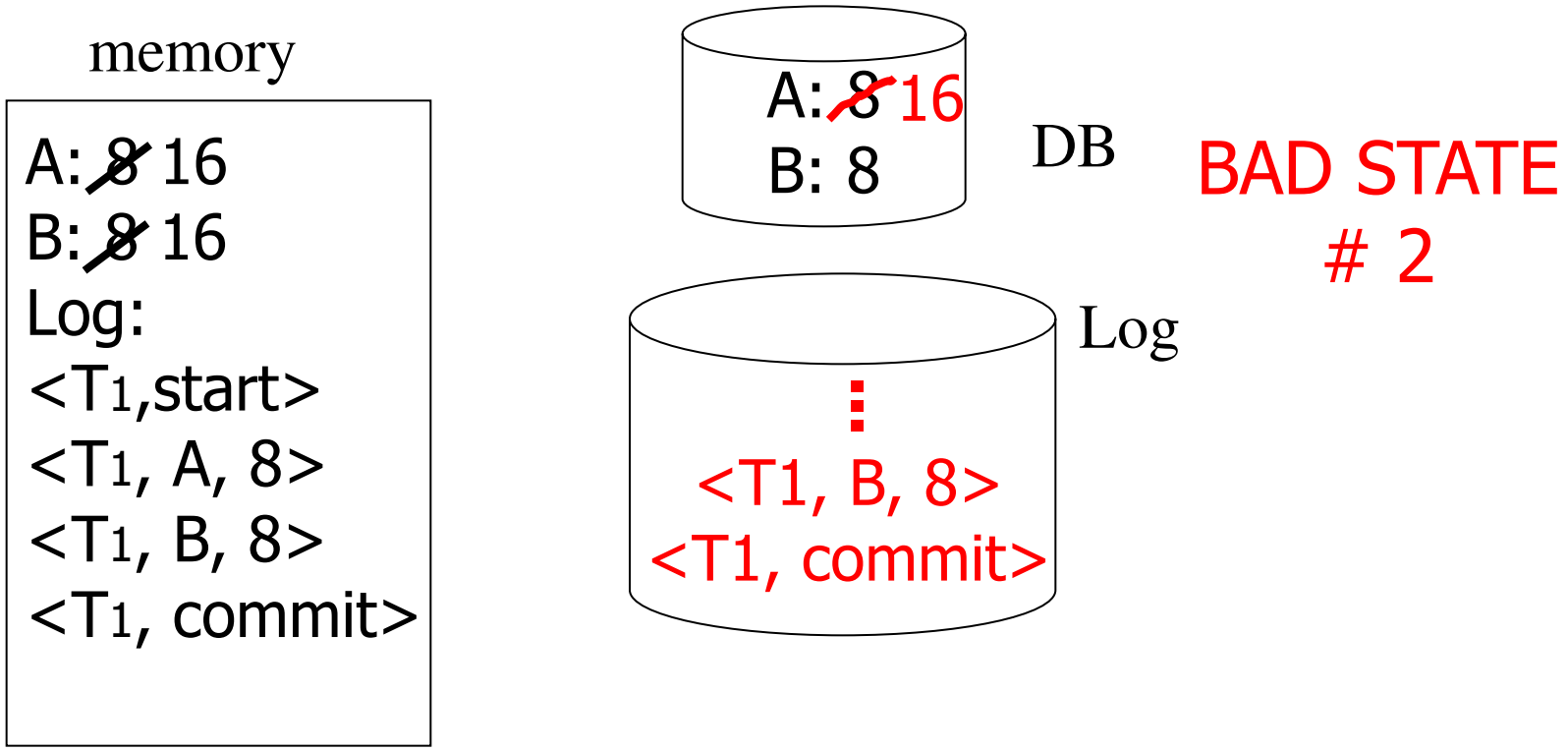
Flushing the Log

- Log is first written in memory
- Not written to disk on every action



Flushing the Log

- Log is first written in memory
- Not written to disk on every action



Recovery rules: Undo logging

- For every Ti with <Ti, start> in log:
  - If <Ti,commit> or <Ti,abort> in log, do nothing
  - Else
    - For all <Ti, *X*, *v*> in log:
      - write *(X, v)*
      - output (*X*)
    - Write <Ti, abort> to log

Recovery rules: Undo logging

(1) Let S = set of transactions with <Ti, start> in log, but no

<Ti, commit> (or <Ti, abort>) record in log

(2) For each <Ti, X, v> in log,

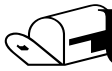
in reverse order (latest $\rightarrow$ earliest) do:

- if Ti $\in$ S then
  - write (X, v)
  - output (X)

(3) For each Ti $\in$ S do

- write <Ti, abort> to log

What if failure during recovery?

No problem!  Undo idempotent

Checkpointing

- Recovery requires that entire log record is examined.
- Deletion of log may not be possible.
- If we truncate the log after the commit of one transaction, log records of other transaction might be lost.
- Solution: checkpoint the log periodically
  - Stop accepting new transactions
  - Wait until all currently active transactions commit.
  - Flush the log to disk.
  - Write the log record <CKPT>, and flush the log again.
  - Resume accepting transactions

Recovery with Checkpoint

- No need to scan beyond check point.

Nonquiescent Checkpointing

- We must shut down the system while checkpoint is made
- Complex technique: nonquiescent Checkpointing
  - Write a log record <START CKPT(T1,…Tk)> and flush the log.
    - T1,..,Tk are active transactions.
  - Wait until all of T1..Tk commit or abort, but do not prohibit other transactions from starting.
  - When all completed, write <END CKPT> record and flush the log.

Recovery with Nonquiescent Checkpointing

- If we first meet the <END CKPT> record, we scan till <START CKP> record and start backwards; previous log is useless.
- If we first meet <START CKPT(T1,..Tk)>, crash occurred during the check point. No need to scan earliest of these transactions.

Example

<START,T1>
<T1,A,5>
<START T2>
<T2, B, 10>

<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>

<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

Outline

- Issues and Models for Resilient Operation
- Undo logging
- **Redo logging**
- Undo/Redo logging
- Protecting against media failures

REDO LOGGING

- Problem with UNDO LOGGING
  - We can not commit without first writing all its changed data to disk.
- We can save disk I/Os if we keep the data as long as we can in the main memory with the ability to fix the things in case of crash.

Differences with redo and undo logging

- During recovery
- 1
  - Undo logging cancels the effect of incomplete transactions and ignores committed ones during recovery.
  - Redo ignores incomplete transactions and repeats the changes made by committed transactions.
- 2
  - UNDO logging requires us to write changed database elements to disk before COMMIT record reaches disk.
  - REDO logging requires that the COMMIT record appear on disk before any changed values reach disk
- 3.
  - To recover, we need new values in REDO logging.

The REDO-logging rule

- Every time T modifies a database element X, a record of the form <T,X,v> must be written to the log.
  - Transaction T wrote a new value of "v" for X. No indication of old values.
- Rule 1
  - Before modifying any data element X on disk, it is necessary that all log records pertaining to this modification of X, including both the update record <T,X,v> and the <COMMIT T> record, must appear on disk.
- Order of operations- REDO logging
  - The log records indicating changed database elements
  - The COMMIT record
  - The changed database elements themselves.

Recovery with REDO

- Identify committed transactions
- Scan the log forward from the beginning. For each log record <T,X,v> encountered.
  - (a) If T is not a committed transaction, do nothing.
  - (b) If T is committed, write value v for X.
- For each incomplete transaction T, write an <ABORT T> record to the log and flush the log.

Checkpointing a REDO log

- Idea: Write all the actions of transactions which have committed and not written to disk i.e. Redo.
- Algorithm:
  - Write a log record <START CKPT<T1,..,Tk>, where T1,..Tk are all active transactionss. FLUSH the log.
  - Write to DISK all database elements that were written to buffers but not yet to disk by transactions that had already committed when the START CKPT record was written to the log.
  - Write <END CKPT> record to the log and
  - FLUSH the log.

Recovery with a Checkpointed Redo Log

- Suppose the last checkpoint record before crash is <END CKPT>.
  - Every transaction that committed before <START CKPT(T1,..,Tk)> has had its changes written to the disk so need not be recovered.
  - Any transaction in the Ti's or started after the beginning of checkpoint can still have changes it made not on disk even if it has committed
  - Perform recovery with redo
  - Start of Ti's can appear prior to any number of checkpoints. Link the START records and redo the transactions.

Recovery with a Checkpointed Redo Log

- Suppose last checkpoint record on the log is a <START CKPT<T1,..,Tk)> record.
  - We are not sure that committed transactions are written to disk.
  - Goto previous <END CKPT> and find its matching <START CKPT (S1, …, Sm)> record
  - REDO all those committed transactions that either started after the START CKPT or are among the Si's
- All the records before the earliest START Ti can be deleted.

Outline

- Issues and Models for Resilient Operation
- Undo logging
- Redo logging
- **Undo/Redo logging**
- Protecting against media failures

UNDO/REDO logging

- UNDO logging requires that data to be written to disk immediately. Increases disk I/Os.
- REDO logging requires us to keep all the modified blocks in buffers until the log records have been flushed. Increases the number of buffers.
- REDO/UNDO logging provides more flexibility,

UNDO/REDO rules

- Log record
  - <T,X,v,w>, T has changed the values of X from v to w.
- UR1
  - Before modifying any database element X on disk because of changes made by T, it is necessary that update record <T,X,v,w> appears on disk.
  - The COMMIT T record can precede or follow.

Recovery with UNDO/REDO Logging

- Rules
  - Redo all the committed transactions in the order earliest-first, and
  - Undo all the incomplete transactions in the order latest-first.

Checkpointing an Undo/Redo log

- It is simpler
  - Write <START CKPT(T1,..Tk)> record in the log, where T1,..Tk are active transactions.
  - Write to disk all the buffers that are dirty.
  - Write <END CKPT> record in the log.

Outline

- Issues and Models for Resilient Operation
- Undo logging
- Redo logging
- Undo/Redo logging
- **Protecting against media failures**

Protecting against media failures

- Log can protect us from system failures when only main memory contents are lost.
- We can also recover when disks crashes if
  - The log is maintained on other disks
  - The log were never thrown away after a checkpoint, and
  - The log is of redo or undo/redo type , so new values are stored on the log.
    - However, it is not practical to keep the entire log, as size of the log becomes very large pretty quickly.

The archive

- Archiving is the solution to media failures
  - Maintain a copy of the database separate from the database it self.
- Shutdown the database and make a copy.
  - Recovery: Backup + Log.
  - Loosing a log
    - Maintain online log at the remote place.
- Writing an archive is a lengthy process.
  - Full dump: Entire database is dumped
  - Incremental dump, only those elements changed since the previous full or incremental dump are copied.
    - Level 0 dump – Full dump
    - Level “i” dump – Copy changes from last dump

Nonquiescient Archiving

- Database is shutdown for a certain period for normal dump.
- Similar to nonquiescient checkpointing we can have nonquiescient archiving.
  - Write a log record <START DUMP>
  - Perform checkpoint
  - Perform full or incremental dump of data.
  - Make sure that log is stored at remote and secure location.
  - Write a log record <END DUMP>

Example

- Say 4 elements A, B, C and D in database with values 1,2,3 and 4
- Dump begins now – Assume database elements copied in order of A, B, C and D
- After dump starts, A changed to 5, C changed to 6 and B changed to 7
- Database at beginning (1,2,3,4)
- Database at end (5,7,6,4)
- However copy of dump (1,2,6,4)

Undo/Redo log for the dump

<START DUMP>

<START CKPT (T1, T2)> (active transactions when dump began)

<T1, A, 1, 5>

<T2, C, 3, 6>

<COMMIT T2>

<T1, B, 2, 7>

<END CKPT>

Dump completes

<END DUMP>

Recovery Using an Archive and log

- Restore the database from archive
  - Find the recent full dump and reconstruct the database from it.
  - If there are incremental dumps, modify the database according to each, earliest first.
  - Modify the database using surviving log.
  - Use the method of recovery appropriate to the logging method.

Recovery

- Suppose failure after <END DUMP> but before <Commit T1>
- Database first restored to (1,2,6,4)
- Looking at log, T2 is committed so redo
- New database (1,2,6,4) since T2 changes C
- T1 did not commit so undo -- restored to (1,2,6,4)