PURDUE UNIVERSITY | Department of Computer Science

CS 44800: Introduction To Relational Database Systems

Failure and Recovery

Prof. Jianguo Wang

11 November 2021

Indiana Center for Database Systems™

PURDUE UNIVERSITY
Department of Computer Science

Integrity or correctness of data

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

EMP

Name	Age
White	52
Green	3421
Gray	1

Integrity or consistency constraints

- Predicates data must satisfy
- Examples:
  - x is key of relation R
  - $x \rightarrow y$ holds in R
  - Domain(x) = {Red, Blue, Green}
  - a is valid index for attribute x of R
  - no employee should make more than twice the average salary

3

Constraints (as we use here) may not capture "full correctness"

Example 1 Transaction constraints

- When salary is updated,

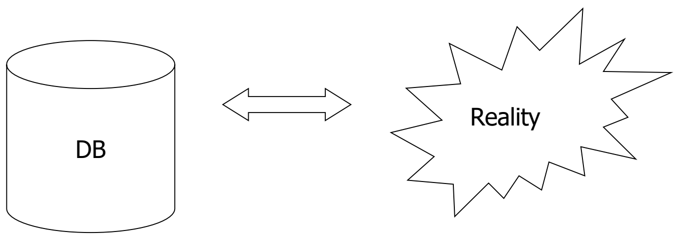
  new salary > old salary
- When account record is deleted,

  balance = 0

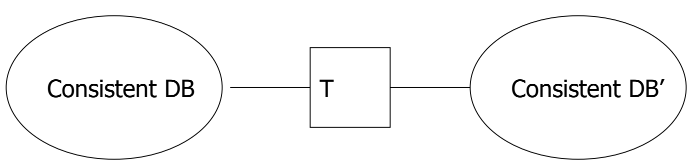
5

Constraints (as we use here) may not capture “full correctness”

Example 2 Database should reflect real world



Transaction: collection of actions that preserve consistency



Big assumption:

If T starts with consistent state + T executes in isolation
$\Rightarrow$ T leaves consistent state

Correctness in the case of failure (informally)

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

Purdue University
Department of Computer Science

How can constraints be violated?

- Transaction bug
- DBMS bug
- Hardware failure
  - e.g., disk crash alters balance of account
- Data sharing
  - e.g.: T1: give 10% raise to programmers
    T2: change programmers $\Rightarrow$ systems analysts

13

Purdue University
Department of Computer Science

Will not consider:

- How to write correct transactions
- How to write correct DBMS
- Constraint checking & repair
  - That is, solutions studied here do not need to know constraints

15

PURDUE UNIVERSITY
Department of Computer Science

Recovery

- First order of business:
  - Failure Model

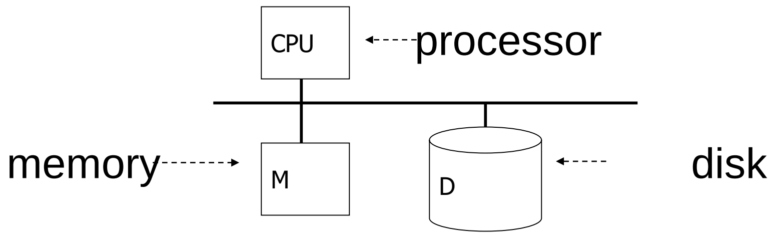
PURDUE UNIVERSITY
Department of Computer Science

Types of Failures

- Events
  - Desired
  - Undesired
    - Expected
    - Unexpected

PURDUE UNIVERSITY
Department of Computer Science

Our failure model



19

PURDUE UNIVERSITY
Department of Computer Science

Desired events: see product manuals….

Undesired expected events:

System crash

- memory lost
- cpu halts, resets

═══ that's it!! ═══

Undesired Unexpected: Everything else!

20

Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

PURDUE UNIVERSITY
Department of Computer Science

Undesired Unexpected: Everything else!

Examples:

- Disk data is lost
- Memory lost without CPU halt
- Sun goes supernova

Department of Computer Science

Is this model reasonable?

Approach: Add low level checks + redundancy to increase probability model holds

E.g.,
- Replicate disk storage (stable store)
- Memory parity
- CPU checks

Storage Structure

- **Volatile storage**:
  - Does not survive system crashes
  - Examples: main memory, cache memory
- **Nonvolatile storage**:
  - Survives system crashes
  - Examples: disk, tape, flash memory, non-volatile RAM
  - But may still fail, losing data
- **Stable storage**:
  - A mythical form of storage that survives all failures
  - Approximated by maintaining multiple copies on distinct nonvolatile media
  - See book for more details on how to implement stable storage

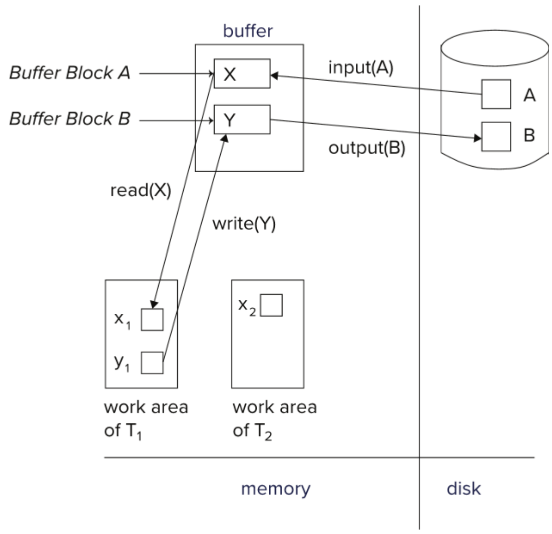
Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

Protecting storage media from failure (Cont.)

- Copies of a block may differ due to failure during output operation.
- To recover from failure:
  1. First find inconsistent blocks:
     1. *Expensive solution*: Compare the two copies of every disk block.
     2. *Better solution*:
        - Record in-progress disk writes on non-volatile storage (Flash, Non-volatile RAM or special area of disk).
        - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
        - Used in hardware RAID systems
  2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

Example of Data Access



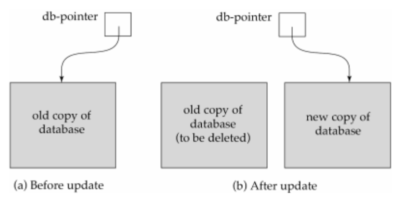
Recovery Algorithms

- Suppose transaction T_i transfers \$50 from account *A* to account *B*
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database.
  - A failure may occur after one of these modifications have been made but before both of them are made.
  - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
  - Not modifying the database may result in lost updates if failure occurs just after transaction commits
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study **log-based recovery mechanisms** in detail
  - We first present key concepts
  - And then present the actual recovery algorithm
- Less used alternative: **shadow-copy** and **shadow-paging** (brief details in book)

shadow-copy



Log-Based Recovery

- A **log** is a sequence of **log records**. The records keep information about update activities on the database.
  - The **log** is kept on stable storage
- When transaction T_i starts, it registers itself by writing a
  <T_i **start**> log record
- *Before* T_i executes **write**(X), a log record
  <T_i, X, V_1, V_2>
  is written, where V_1 is the value of X before the write (the **old value**)**,** and V_2 is the value to be written to X (the **new value**).
- When T_i finishes it last statement, the log record <T_i **commi**t> is written.
- Two approaches using logs
  - Immediate database modification
  - Deferred database modification.

Immediate Database Modification

- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - (Will see later that how to postpone log record output to some extent)
- Output of updated blocks to disk can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.
- The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit
  - Simplifies some aspects of recovery
  - But has overhead of storing local copy

Transaction Commit

- A transaction is said to have committed when its commit log record is output to stable storage
  - All previous log records of the transaction must have been output already
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

Immediate Database Modification Example

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_0, B, 2000, 2050>		
	A = 950	
	B = 2050	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B , B_C
<T_1 **commit**>		
		B_A

B_C output before T_1 commits

- Note: B_X denotes block containing X.

B_A output after T_0 commits

34

PURDUE UNIVERSITY
Department of Computer Science

Recovery

- Previous example – what happens if failure in between (or during) some of the writes
  – Commit hasn't been written, so transaction hasn't happened
- *Undo* logging *(Ariadne ~900BCE)*
  – Restore state to before uncommitted transaction started
  – Log contains information on old state
  – "Follow the string" back to before you started

35

PURDUE UNIVERSITY
Department of Computer Science

Write-Ahead Logging

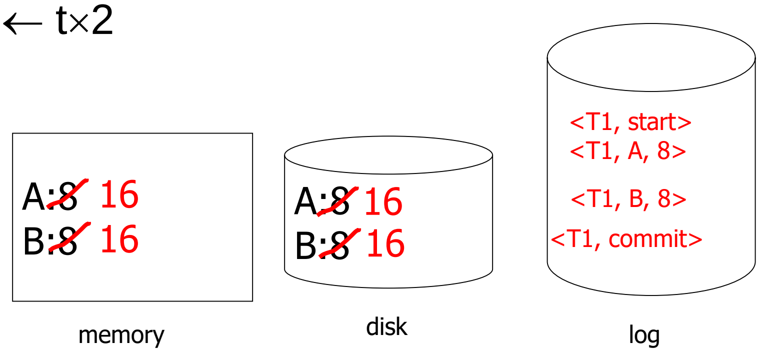
- Logging must happen before write
  - Log captures state before write
  - Undo either restores written block to previous state, or contains same value if write hasn't occurred
- When commit happens
  - All blocks modified by transaction
  - Commit T_i written to log
- When undoing
  - If commit T_i in log, then ignore log entries for T_i

36

PURDUE UNIVERSITY
Department of Computer Science

Undo logging (Immediate modification)

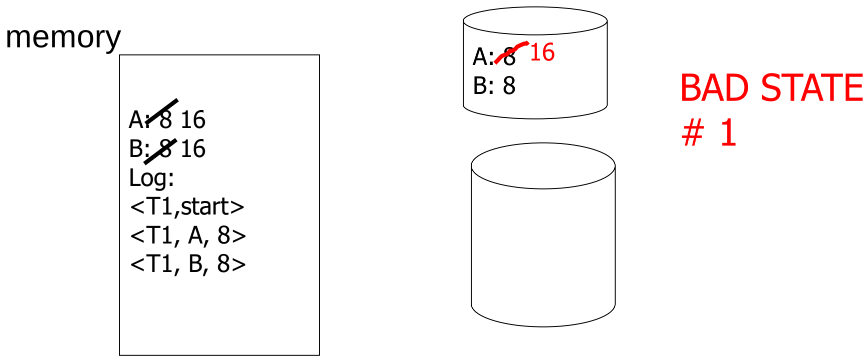
T1: Read (A,t); t ← t×2   A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);



37

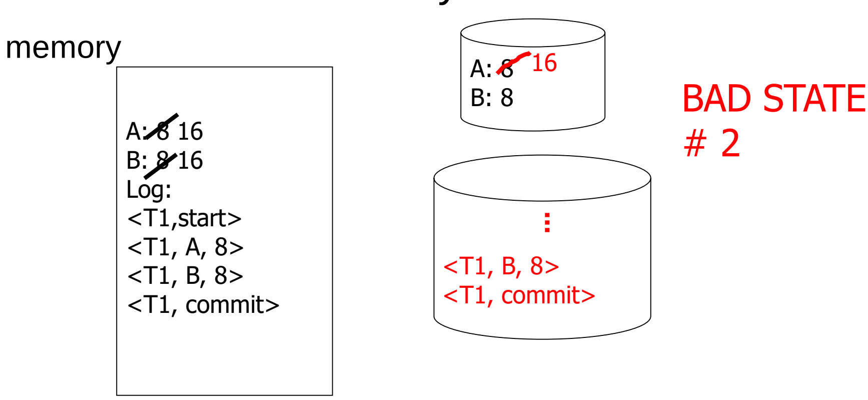
One "complication"

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



One "complication"

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



PURDUE UNIVERSITY
Department of Computer Science

Undo logging rules

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

PURDUE UNIVERSITY
Department of Computer Science

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

⊠IS THIS CORRECT??

PURDUE UNIVERSITY
Department of Computer Science

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

42

Concurrency Control and Recovery

- With concurrent transactions, all transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions
- We assume that *if a transaction T_i has modified an item, no other transaction can modify the same item until T_i has committed or aborted*
  - i.e., the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise, how to perform undo if T_1 updates A, then T_2 updates A and commits, and finally T_1 has to abort?
  - Can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (strict two-phase locking)
- Log records of different transactions may be interspersed in the log.

44

Database System Concepts - 7th Edition 19.44 ©Silberschatz, Korth and Sudarshan

Undo and Redo Operations

- **Undo and Redo of Transactions**
  - **undo**(T_i) -- restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
    - Each time a data item X is restored to its old value V a special log record <T_i , *X, V*> is written out
    - When undo of a transaction is complete, a log record <T_i **abort**> is written out.
  - **redo**(T_i) -- sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
    - No logging is done in this case

Recovering from Failure

- When recovering after failure:
  - Transaction T_i needs to be undone if the log
    - Contains the record <T_i **start**>,
    - But does not contain either the record <T_i **commit**> *or* <T_i **abort**>.
  - Transaction T_i needs to be redone if the log
    - Contains the records <T_i **start**>
    - And contains the record <T_i **commit**> *or* <T_i **abort**>

Recovering from Failure (Cont.)

- Suppose that transaction T_i was undone earlier and the <T_i **abort**> record was written to the log, and then a failure occurs,
- On recovery from failure transaction T_i is redone
  - Such a **redo** redoes all the original actions of transaction T_i *including the steps that restored old values*
    - Known as **repeating history**
    - Seems wasteful, but simplifies recovery greatly

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000, and log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> are written out

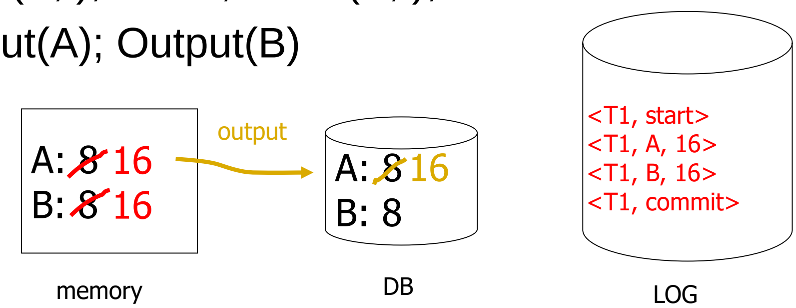
(b) redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and C is restored to 700. Log records <T_1, C, 700>, <T_1, **abort**> are written out.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

PURDUE UNIVERSITY
Department of Computer Science

Redo logging (deferred modification)

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



49

PURDUE UNIVERSITY
Department of Computer Science

Redo logging rules

1. For every action, generate redo log record (containing new value)
2. Before X is modified on disk (DB), all log records for transaction that modified X (including commit) must be on disk
3. Flush log at commit

50

Recovery rules: Redo logging

- For every Ti with <Ti, commit> in log:
  - For all <Ti, X, v> in log:

    { Write(X, v)
      Output(X) }

⊠IS THIS CORRECT??

Recovery rules: Redo logging

1. Let S = set of transactions with <Ti, commit> in log
2. For each <Ti, X, v> in log, in forward order (earliest → latest) do:
   - if Ti ∈ S then { Write(X, v)
     Output(X) } ← optional