Transaction Management

Recovery (1)

Review: ACID Properties

- Atomicity
  - Actions are never left partially executed
- Consistency
  - Actions leave the DB in a consistent state
- Isolation
  - Actions are not affected by other concurrent actions
- Durability
  - Effects of completed actions are resilient against system failures

Integrity or Correctness of Data

A
Consistency
I
D

- Data should be “accurate” or “correct” at all times
- Examples:

Name	Age
While	52
Green	3421
Gray	1

Integrity or Consistency Constraints

- Data should satisfy the predicates
- Examples:
  - The funds that Ada can transfer from her saving account to her chequing account should not exceed the total amount in her saving account.

Definition

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

- Transaction: collection of actions that preserve consistency

Consistent DB

Transaction T

Consistent DB’

Big Assumption

If T starts with consistent state +
T executes in isolation

↓

T leaves consistent state

Correctness (informally)

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

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 → systems analysts

How Can We Prevent/Fix Violations?

- Chapter 17: due to failures only
- Chapter 18: due to data sharing only
- Chapter 19: due to failures and sharing

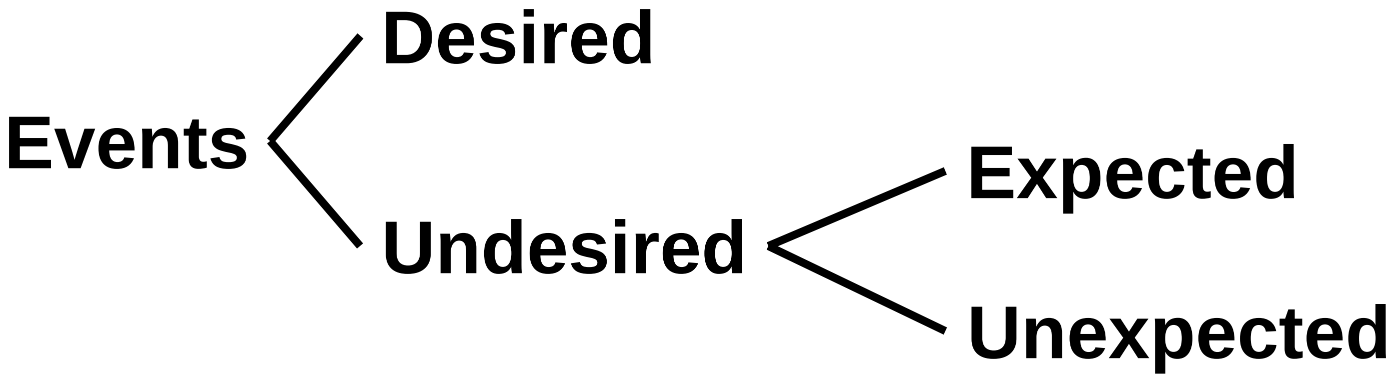
Will Not Consider

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

Chapter 17: Recovery

- First order of business:

Failure Model



Events

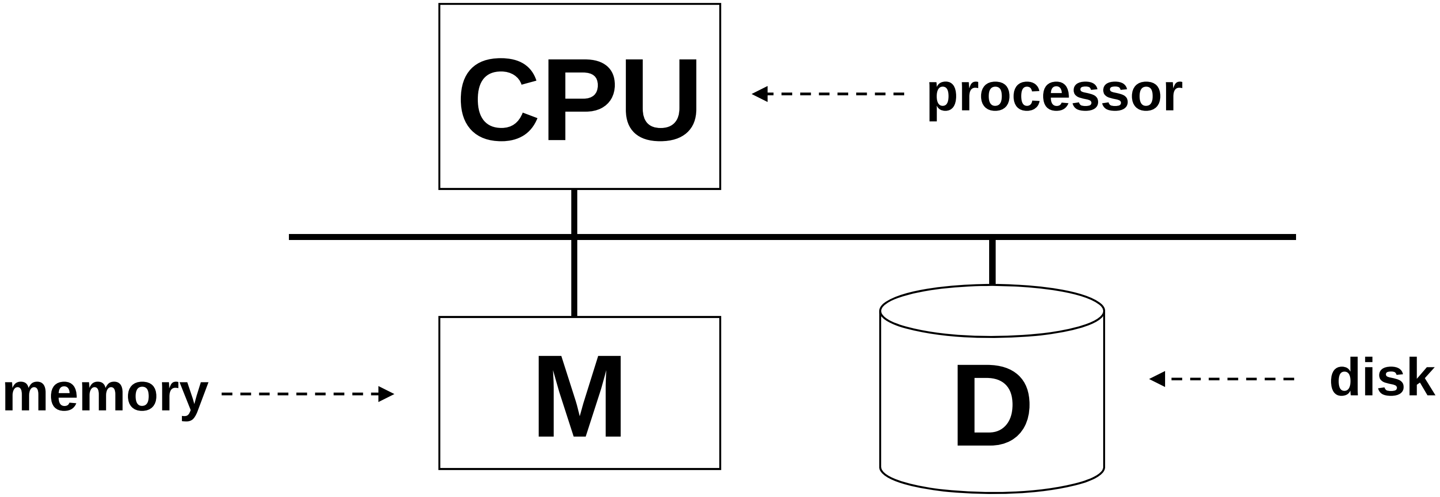
Desired

Undesired

Expected

Unexpected

Our Failure Model



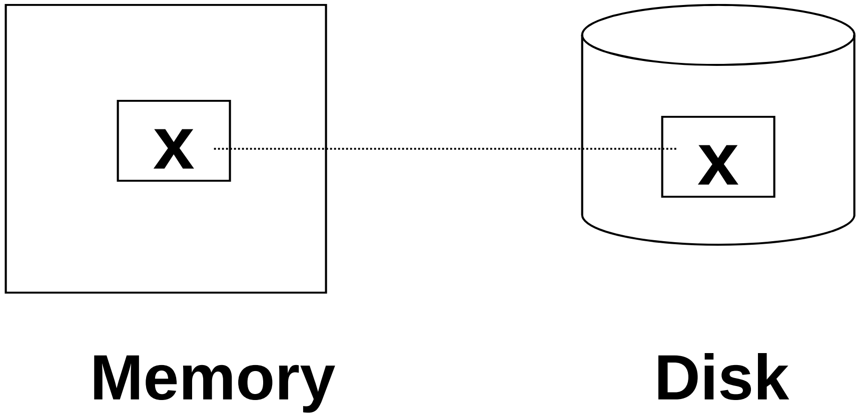
Events

- Desired events:
  - See product manuals…
- Undesired expected events:
  - System crash
    - Memory lost
    - CPU halts, rests
- Undesired unexpected events:
  - Everything else!
    - Disk data is lost
    - Memory lost without CPU halt
    - CPU catches fire
    - Car smashes into data center
    - (You get the idea…)

Data Storage

- Second order of business:

Storage Hierarchy



Operations

- Input (x): block containing x → memory
- Output (x): block containing x → disk

- Read (x,t): do input(x) if necessary
  t ← value of x in block
- Write (x,t): do input(x) if necessary
  value of x in block ← t

Failure Models

- Undesired expected:
  - System crash
    - Data on disk still there on restart
- Undesired unexpected:
  - Media failure
  - Catastrophic failure
    - Data on disks lost!

System Crash

A
C
I
Durability

- Problem 1:
  - Transaction completed, but results only in memory
- If system crashes, effects of transaction lost
- Solution: don’t tell user the transaction has “committed” until all effects are on disk!

System Crash

- Problem 2:
  - Unfinished transaction
- Example:

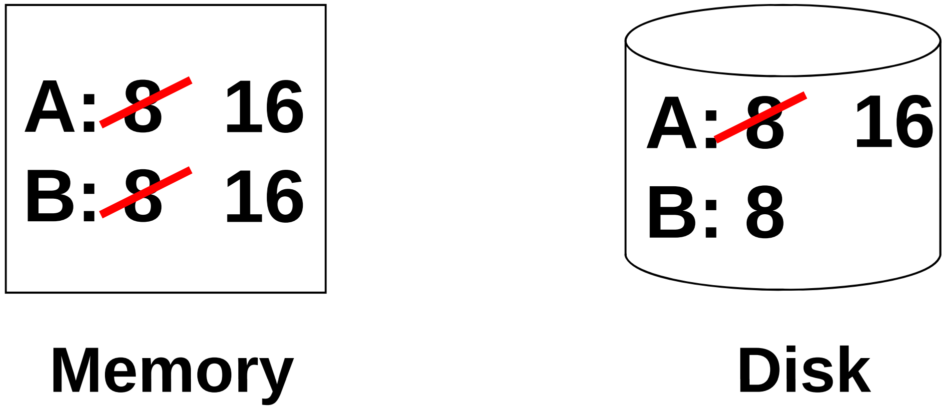
Constraint: A=B

$$T_1: A \leftarrow A \times 2$$
$$B \leftarrow B \times 2$$

Example

T1:
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);

Crash!


A: ~~8~~ 16
B: ~~8~~ 16

A: ~~8~~ 16
B: 8

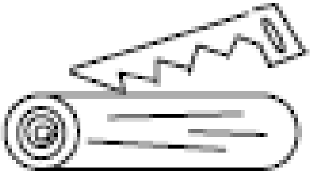
Memory

Disk

Solution

Atomicity
C
I
D

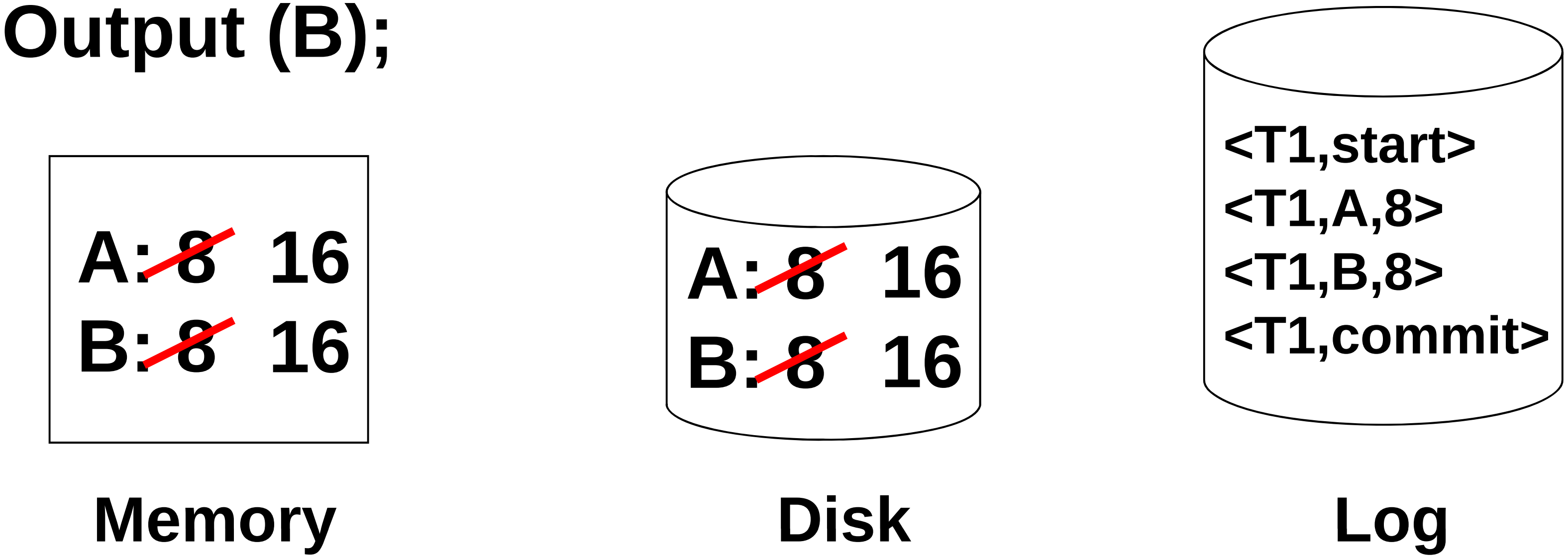
- Need atomicity:
  - Execute all actions of a transaction or none at all
- Solution: logging



One Variation: Undo Logging

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);

Immediate Modification



A: ~~8~~ 16
B: ~~8~~ 16

Memory

A: ~~8~~ 16
B: ~~8~~ 16

Disk

<T1,start>
<T1,A,8>
<T1,B,8>
<T1,commit>

Log

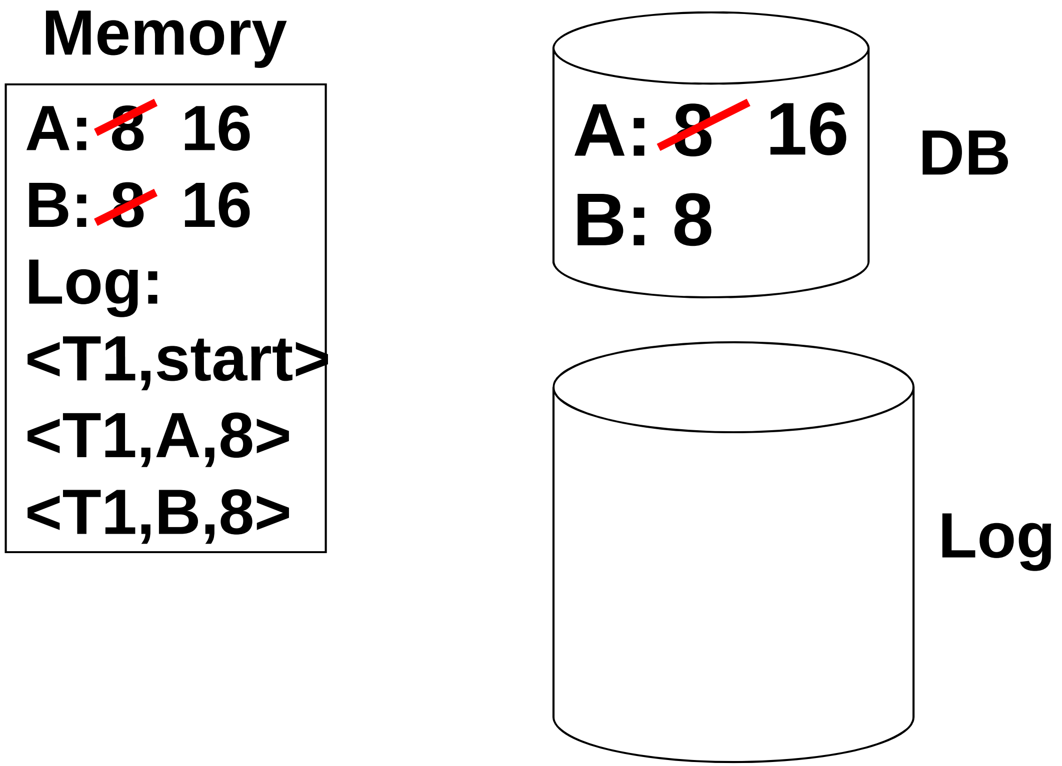
Assumption

- Logging “fakes” atomicity by undoing writes of partially completed actions.

- Assumption: writing a block is atomic
  - Atomicity of block writes is itself “faked” by low-level checks
  - And so it goes…
  - At the bottom, there must be some intrinsically atomic action (writing a bit?)

One “Complication”

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



Memory

A: ~~8~~ 16
B: ~~8~~ 16
Log:
<T1,start>
<T1,A,8>
<T1,B,8>

A: ~~8~~ 16
B: 8

DB

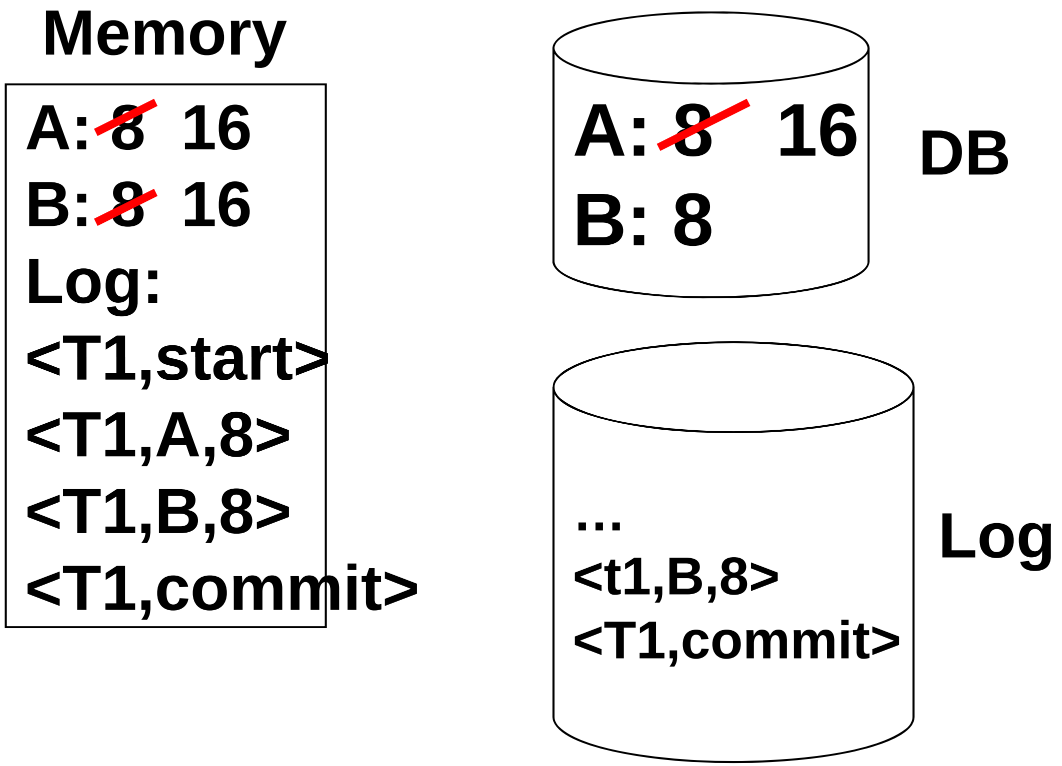
Log

Bad State 1

Data has written to disk, but log has not.

One “Complication”

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



Memory

A: ~~8~~ 16
B: ~~8~~ 16
Log:
<T1,start>
<T1,A,8>
<T1,B,8>
<T1,commit>

A: ~~8~~ 16
B: 8

DB

…
<t1,B,8>
<T1,commit>

Log

Bad State 2

Commit log has written to disk, but data has not.

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.

(3) Before commit is flushed to log, all writes of transaction must be reflected on disk.

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?

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 → earliest) do:

    - if Ti ∈ S then
        - write (X, v)
        - output (X)

(3) For each Ti ∈ S do

    - write <Ti, abort> to log

What if failure during recovery?
No problem! Undo idempotent.

To Discuss

- Redo logging
- Undo/redo logging, why both?
- Real work actions
- Checkpoints
- Media failures
- Catastrophic failures